DIPLOMAMUNKA

Bodogan Judit Mária

Debrecen

2008

Debreceni Egyetem
Informatika Kar

Tesztelési eszközök és módszerek

Témavezető:
Dr. Juhász István
Egyetemi Adjunktus

Szerző:
Bodogan Judit Mária
Programtervező Matematikus

Debrecen
2008

Tartalomjegyzék

Ábrajegyzék

1 Bevezetés

1.1 Diplomamunkám témája

Diplomamunkám során az alkalmazások teszteléséről írok. Bemutatom a tesztelés fázisait, különböző tesztelési módszereket és eszközöket, melyek megkönnyítik a tesztelést. A dolgozatomba az eszközök többségének működését konkrét példákon keresztül szemléltetem.

1.2 Témaválasztás

Diplomamunkám témájának választását több dolog is befolyásolta. Az egyik ok, amiért ezt választottam, az hogy az egyetemi képzésem során kevés olyan tantárggyal találkoztam, amely legalább elméleti szinten kitér az alkalmazások tesztelésére. Ezen nem kell elcsodálkoznunk, hiszen napjainkban is akadnak olyan szoftver gyártó cégek, ahol a tesztelés nem kapja meg a megfelelő figyelmet. Így a tesztelések ad-hoc módon folynak Ezek a cégek a tesztelésen próbálnak spórolni, nem gondolván ennek súlyos következményeire. Ezáltal a szoftver minősége gyenge lesz, a karbantartási költségei megnőnek.
A másik ok az, hogy sokan úgy vélik, hogy a teszteléshez nem szükséges szakértelem. A legtöbb ember nem tudja miből is áll egy tesztelő munkája. Akadnak olyan velem egy szakon végzett hallgatók is, akik azt hiszik, hogy a tesztelő egész nap véletlenszerűen kattintgat egy felületen.
Diplomamunkámmal, melyben részletesen leírom a tesztelés módszereit és fázisait, azt szeretném megmutatni, hogy a tesztelők munkája nem olyan egyszerű, mint azt egyesek hiszik, mert gondoljunk csak bele, hogy nekik kell a legjobban átlátniuk és megérteniük a szoftver működését. Másik cél, hogy rávilágítsak a tesztelés fontosságára a szoftver fejlesztés során. Az elkészített szoftvereknek szigorú követelményeknek kell eleget tenniük.

2 Követelmények meghatározása és elemzése

A rendszereink nagyok, komplexek, nehezen lehet őket körülhatárolni, nehéz előre megmondani, hogy hogyan, milyen körülmények között, mit szolgáltatva működjön. A rendszer szolgáltatásainak, és az ezekre vonatkozó megszorításoknak az együttesét hívjuk követelményeknek. Az egész folyamat a követelmények megtalálásának, adott esetben kitalálásának, elemzésének, dokumentálásának és validálásának lépésével indul. Ezt hívjuk a követelmények meghatározásának és elemzésének, vagy szokás ezt követelménytervezésnek hívni.

A követelmények osztályozása:

- *felhasználói követelmények*: azt specifikálják, hogy a rendszernek milyen szolgáltatásai legyenek, és ezek hogyan, milyen körülmények között, milyen elvárásokkal működjenek. A felhasználói követelmények tehát a funkcionalitást jelentik. Például egy banki rendszertől követelményként elvárjuk, hogy lehessen benne ügyfeleket, folyószámlákat kezelni, szerződéseket rögzíteni, átutalásokat végrehajtani, ügyfelek adatait, egyenlegeit lekérdezni és még egyéb dolgokat. Elvárjuk, hogy a rendszer automatikusan tudja kezelni ezeket a fogalmakat
- *rendszerkövetelmények*: azok a követelmények, amelyek általában a rendszer teljes egészére vonatkoznak, melyek függetlenek az egyes funkcióktól, szolgáltatásoktól. Szoktak belső követelményekről beszélni, például hozzáférhetőség, válasz idő, tár szükséglet. A rendelkezésre állás nyilván független attól, hogy milyen szolgáltatást nyújt a rendszer, el kell tudni érni éjjel-nappal. Egy banki rendszertől azt várjuk el, hogy az év minden percében érzékelje az ügyfelek számláin történő változásokat. Természetesen a belső követelményeken túl lehetnek egyéb speciális rendszerkövetelmények
- *rendszer-specifikáció*: tartalmazza a rendszer fejlesztéséhez szükséges struktúrát, képernyő terveket, adatbázis tervezést, valamint a folyamatok leírását. A specifikáció tulajdonképpen a koncepció terv technikai megfogalmazása. Ez általában kiegészítése a

rendszerkövetelmény specifikációnak, ez a kifejlesztendő szoftvernek egy absztrakt specifikációja, amely majd a tervezési fázis alapjait szolgáltatja

A felhasználói követelmények azok a követelmények, amelyekkel elsősorban a fejlesztői oldali és a megrendelő oldali menedzserek találkoznak. Nagyon lényeges, hogy a felhasználói követelmények elsőrendű fontosságúak a végfelhasználók számára. Tehát egy banki rendszertől felhasználói szinten elvárjuk, hogy az összes banki dolgozó, mint végfelhasználó kívánalmait, megszorításait tudja teljesíteni.
A rendszerkövetelményekkel a rendszerfejlesztők, az informatikusok találkoznak, mivel nekik kell olyan rendszert fejleszteni, amely a rendszerkövetelményeknek eleget tesz. A másik oldalon az üzemeltetőknél a rendszergazdák, a rendszeradminisztrátorok számára elsődlegesek. Mert például nekik kell konfigurálni a rendszert.
A szoftverterv specifikáció nyilvánvalóan a szoftver tervezőinek alapvető fontosságú, befolyásolja a fejlesztő munkáját is.
Az alkalmazás fejlesztés során elkészített UML diagramok a rendszerrel szembeni követelményeket tartalmazza. A *usacase* diagramok a fizikai terv részei.
A használati eset diagram segítségével egyértelművé válnak a rendszer határai, a rendszert használók szerepkörei, és az általuk elérhető funkciók. A programozók számára pedig, egyértelmű, hogy milyen funkciókat kell megvalósítani. Ezek a felhasználói esetek a fejlesztés menetének ütemezésére is felhasználhatóak. A legfontosabb funkciókat kijelölve és először azokat fejlesztve, azokat több ideig lehet tesztelni, így azok megbízhatóbbak lesznek. A használati eset diagram tekinthető az alkalmazással szemben támasztott követelmények térképeként, a funkciók grafikus tartalomjegyzékként.

3 Hibamodell és alapfogalmak

A szoftverhibák alapvető sajátsága, hogy a működés teljes időtartama alatt jelen vannak. A kérdés az, hogy a hatásuk miként nyilvánul meg a különböző szituációkban. A hibáknak két alaposztályát különböztetjük meg:

- *Specifikációs hibák:* Azok a hibák, amelyek a fejlesztési ciklus kezdetén képződnek, és a szoftver téves működésében nyilvánulnak meg azáltal, hogy nem teljesülnek a valós felhasználói követelmények. A téves működés tág értelemben tekintendő: Egyaránt következménye lehet a hibás, a hiányos, valamint a következetlen, ellentmondást tartalmazó specifikálásnak. Ezt a kategóriát nevezhetjük még *külső* vagy *felhasználói hibának* is
- *Programozási hibák:* Azoknak a hibáknak a széles köre tartozik ide, amelyeket a programozók követnek el, az előzőleg már specifikált szoftver tervezési és kódolási folyamatában. Ennek a kategóriának másik szinonimái: *belső* vagy f*ejlesztési* hibák. Néhány lehetséges hibatípust ezekből az alábbiakban sorolunk fel:
    - hibás funkcióteljesítés
    - hiányzó funkciók
    - adatkezelési hibák az adatbázis elérése során
    - kezdési és befejezési hibák
    - hibák a felhasználói interfészben
    - határértékek alá vagy fölé kerülés
    - kódolási hiba
    - algoritmikus hiba
    - inicializálási hiba
    - a vezérlési folyamat hibája
    - adatátviteli hiba
    - input-output hiba
    - program blokkok közötti versenyhelyzet
    - programterhelési hiba

- *Tesztelési hibák:* ilyenkor a tesztelt szoftvernek a tesztelés ideje alatt tapasztalt hibás viselkedését nem a szoftver hibái okozzák. A tesztelési hibák lehetnek:
    - teszt specifikációs vagy tesztadat hiba: a hibát ilyenkor a vizsgálati pont minősítésének hibás specifikációja, vagy a rosszul megválasztott tesztadatok okozzák
    - a teszt hibás végrehajtása: a hibát ilyenkor a teszt nem megfelelő végrehajtása (például a végrehajtás lépéseinek helytelen sorrendje) okozzák
    - hibás tesztkörnyezet: a hibát ilyenkor a tesztkörnyezet hibás megválasztása (például hibás tesztprogram) okozza

A tesztelési hibák feltárásakor, a hibát okozó elemet el kell hárítani (például javítani kell a teszt specifikációt, módosítani kell a teszt adatokat, a tesztkörnyezetet, a teszt lépéseinek végrehajtási sorrendjét), majd meg kell ismételni a tesztet.

A gyakorlati tapasztalatok alapján állítható, hogy a teljes fejlesztési költségek mintegy 50%-át teszik ki a tesztelési költségek. Ez magában foglalja a tesztelési folyamatok megtervezését és végrehajtását. A szoftver-technológia fejlődésével a rendszerek mérete és bonyolultsága egyre növekszik. Ennek következményeként állandó igény mutatkozik arra, hogy újabb és hatékonyabb tesztelési módszereket és eszközöket dolgozzunk ki.

A szoftverfejlesztésben fontos követelmény a teljes folyamat konzisztens módon való végig vitele. Az egyes fejlesztési állomások eredményei saját megjelenési formával rendelkeznek. A folyamat helyes végig vitele megköveteli az egyes reprezentációk közötti összhang bizonyítását. Azt kell bizonyítani, hogy a végső szoftver termék 100%-ig megfelel a kiindulási specifikációnak. Ennek a bizonyítási folyamatnak a megvalósítása oly módon történik, hogy az egymást közvetlenül követő fejlesztési fázisok közötti összhangot bizonyítjuk lépésenként. Ha egy fázisnál lényeges különbség mutatkozik, akkor azt addig kell módosítani, amíg az nem harmonizál az előző fázissal. A leírt tevékenységet, amelyben két egymást követő fázis közötti ekvivalenciát bizonyítjuk, verifikációnak nevezzük.

A *verifikáció* definíciója: a *verifikáció* az a folyamat, amelyben igazoljuk, hogy a szoftver egy fejlesztési fázisban teljesíti mindazokat a követelményeket, amelyeket az előző fázisban specifikáltunk.

A lépésenkénti verifikálás elvileg elegendő az ekvivalencia igazolására a kiindulási és a befejező fázis között. Mindazonáltal, mivel a teljes folyamat általában nem egzakt, vagyis nem biztosít abszolút bizonyítást egyik lépésben sem, egy külön önálló *végső verifikációra* is szükség van, amit a specifikáció és a végtermék között hajtunk végre.

Másfelől nézve, ennél a pontnál meg kell jegyeznünk, hogy a verifikációs folyamat tökéletes megvalósítása sem garantálná a végtermék tökéletes használhatóságát. Mindaz, amit garantálni lehet, a kiindulási specifikációval való ekvivalencia teljesülése. Abban az esetben, ha hiba volt a specifikációban, a végtermék nem fogja kielégíteni a felhasználói követelményeket. Emiatt szükség van még egy külön vizsgálati folyamatra is, amiben a terméket az eredeti rendeltetése szempontjából ellenőrizzük. Az ilyen jellegű bizonyítási folyamatot *validációnak* nevezzük.

A *validáció* definíciója: a *validáció* a szoftvernek olyan vizsgálata és kiértékelése, amiben meghatározzuk, hogy minden szempontból teljesíti-e a felhasználói követelményeket.

Egy biztonságkritikus rendszer esetében a következőket kell bizonyítani:

- funkcionálisan megfelelő működés
- megfelelő teljesítmény.
- a biztonsági követelmények kielégítése.

A tökéletlen specifikálás következményeként leginkább a biztonság lesz veszélyeztetve. Mindezek után, a végső validációs eljárásban azt kell eldönteni, hogy a teljes rendszer biztonságos-e vagy sem. Ha valamilyen probléma adódik, akkor vissza kell térni a kezdeti specifikációhoz, és módosítani kell azt. A módosítás azzal jár, hogy újra kell tervezni a rendszert, a szükséges változtatások végig vitelével, minden egyes verifikációs fázist elvégezve, másrészt új validálásra is sort kell keríteni.

A verifikáció annak ellenőrzése, hogy a program megfelel-e a specifikációjának. A validáció pedig annak eldöntésére irányul, hogy a megvalósított program teljesíti-e a felhasználó elvárásait.

Az első kategória a nem megfelelő fejlesztési folyamat következménye, a második pedig, a végső felhasználási követelményekkel való összhang hiánya. A következőkben ezt a két kategóriát rendre *fejlesztési hibának*, illetve *felhasználói hibának* fogjuk nevezni.

3.1 A tesztelés stratégiai terv

A tesztelés célja az alkalmazás fejlesztése során a technikai és felhasználói szintű működés helyességének ellenőrzése, a hibák kiszűrése, és a rendszer működési peremfeltételeinek meghatározása. A sikeres tesztelés eredményeképp előálló rendszer összhangban van a hozzá kapcsolódó műszaki és felhasználói dokumentációval, illetve a követelményspecifikációban megfogalmazott funkcionális követelményekkel, és azoknak megfelelően, megbízhatóan működik.

A tesztelés stratégiai terv célja az alkalmazással kapcsolatos tesztelési tevékenységek összehangolása, a tesztelési stratégia kialakítása, a szükséges teszttípusok, illetve azok hatókörének azonosítása, valamint a tesztelésre használt eszközök és a tesztelésben résztvevő szerepkörök meghatározása.

A tesztelés stratégia terv a hozzá kapcsolódó tesztdokumentációk központi dokumentuma. Az előre való tervezés lényeges mind a költségek becslése, mind pedig minimalizálása szempontjából.

3.2 Tesztkövetelmények

A sikeres teszteléshez különböző követelmények teljesülése szükséges. Ezek az alábbiak:

- funkcionális követelmények, illetve használatieset-diagramok és leírások rendelkezésre állása. Ezen dokumentációk alapján határozhatóak meg a tesztelési forgatókönyvek, illetve azok különböző kimenetelei, a sikeresség vagy sikertelenség tényének megállapítására szolgáló információk.
- rendszerkövetelmények rendelkezésre állása: a terheléses tesztet a rendszerkövetelményekben megadott konfigurációval rendelkező számítógépeken is végre kell hajtani
- felhasználói, telepítési és üzemeltetési kézikönyvek rendelkezésre állása: szintén a tesztelési forgatókönyvek meghatározásához szükségesek (a funkcionális, felületi, a telepítési, a konfigurációs, a migrációs tesztekhez egyaránt)

- projektterv rendelkezésre állása: a tesztelés ütemezéséhez és erőforrás-kezeléséhez szükséges információk

Néhány tesztelési aranyszabály, amit jó szem előtt tartani:

- elvárt eredményt mindig specifikáljunk
- programozó ne tesztelje a saját programját
- minden teszteset eredményét ellenőrizni kell
- kivételes viselkedést is teszteljünk
- azt is igazoljuk, hogy egy program nem úgy működik, ahogy nem kéne
- a teszteseteket meg kell tudni ismételni
- soha ne feltételezzük azt, hogy a program hibátlan. Egy programban mindig vannak hibák, ezek sokszor csoportosan jelentkeznek
- a tesztelés célja a hibák megtalálása (a jó tesztadat az, ami ezt előhozza)

4 Egységek, modulok tesztelése

A moduláris fejlesztés lehetővé teszi a moduláris vagy egységenkénti tesztelést. Ez a fejlesztés közbeni legalacsonyabb szintű tesztelés. Szokás ezt a fajta tesztelést, fehér-doboz tesztelésnek is nevezni (white-box), mivel a tesztelés során bele látunk a rendszer belső működésébe.

Ezen tesztek célja a tesztelendő komponens minél alaposabb "megmozgatása" az összeépítés elõtt. A megmozgatás a programmodul nyilvános és kevésbé nyilvános metódusainak alapos végighívogatását jelenti. Előfordulhat, hogy az alapos végighívogatáshoz nem elegendőek a nyilvános metódusok, ez esetben az egységteszt céljára külön tesztmetódusokat szokás nyitni, amelyek az egység nehezen elérhető részeihez engednek hozzáférést.

A tesztek sikerességének feltétele, az hogy minden teszt sikeres legyen, ellenkező esetben a fejlesztők minden sikertelen tesztről megállapítják, hogy a feltárt hiba nem kritikus, a fejlesztés előrehaladását nem befolyásolja.

A tesztelés e fázisában is készülnek tesztesetek, ezeket nem csak egyszer használatosak, mindannyiszor végre kell hajtani őket ahányszor a rendszeren változást, hajtanak végre.

A modul tesztek során teszteljük az adott modult és a modult a rendszerhez csatoló interfész funkcionális működését.

Az egységtesztek a hibák nagy részét feltárják, létrehozásuk, karbantartásuk könnyebb, kényelmesebb, mint a későbbi fázisokban létrehozott teszteké. Egy hiba minél később kerül felderítésre, annál magasabb a javítási költsége.

A tesztet egy erre a célra létrehozott adatbázison végezzük, melyet a teszt után mindig visszaállítunk eredeti állapotába.

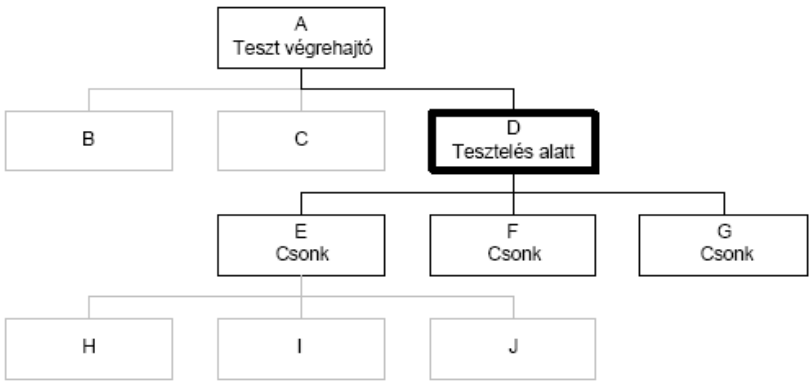
Az egységtesztelés szervezésénél három stratégiát említhetünk meg:

- *Izolációs tesztelés:* a modul az általa hívott és az őt hívó moduloktól elszigetelten kerül tesztelésre. Minden modulhoz szükség van egy tesztvégrehajtó komponensre és az összes általa hívott egységeket helyettesítő csonkokra. Az egységek tetszőleges sorrendben tesztelhetők, mivel egyik egység teszteléshez sincs szükség másik már tesztelt modulra.

  Előnyök: egyszerű tesztvégrehajtó egységek, az egységek párhuzamosan tesztelhetők, a módosítások csak az adott egység változását vonják maguk után.

  Hátrányok: nem biztosítják az egység korai integrációját.

  (Ez a legjobb választás egységtesztelésre)



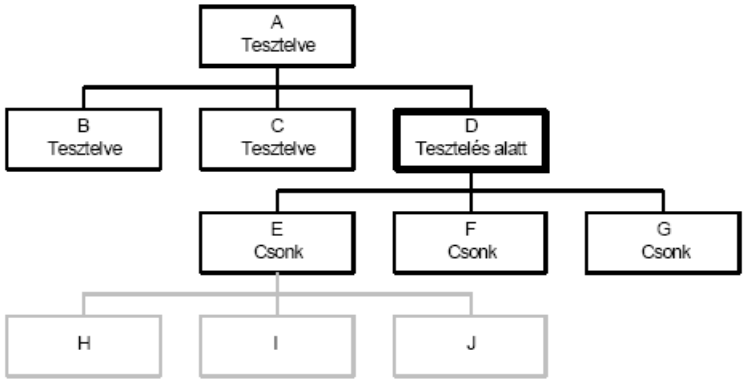
1. ábra Izolációs tesztelés

- *Top-down tesztelés:* először a hierarchia csúcsán álló modul kerül tesztelésre. Az általa hívott modulokat csonkok helyettesítik. Ezt addig ismételjük, amíg a legalsó szinten levő egységek is tesztelésre kerülnek.

  Előnyök: biztosítja a szoftver integrációs fázis előtti integrációját, ezzel azonosítható az alsóbb szintű egységekben megvalósított redundáns funkcionalitás

  Hátrányok: a tesztesetek több csonkot használhatnak egyszerre, a tesztelés egyre komplikáltabb lesz, a módosítás befolyásolhatja a hierarchiában egy szinten vagy alacsonyabban levő egységek tesztelését.

  (A már tesztelt egységek integrálásának eszköze)



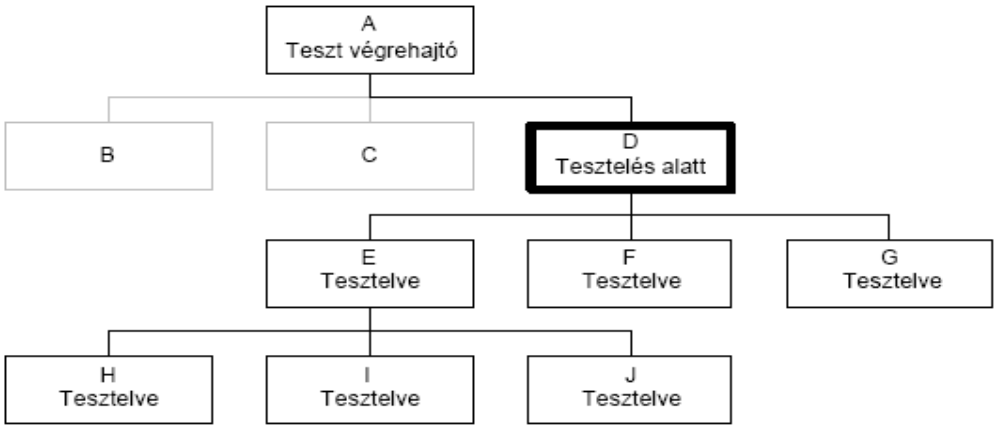
2. ábra Top-down tesztelés

- *Bottom-up tesztelés:* először a legalsó szinten lévő modulokat kell tesztelni, majd ezeket kell használni a felsőbb szintű egységek tesztjeinél. A modulok az őket hívó moduloktól elszigetelten kerülnek tesztelésre, de saját hívásukhoz a valódi hívott modulokat használja.

  Előnyök: nincs szükség csonkokra, biztosítja az egységek korai integrációját, a hierarchia alján lévő modulok tesztelése egyszerű, jól használható objektumok tesztelésére

  Hátrányok: egyre bonyolultabb, egyre nehezebben karbantartható a tesztelés, ahogy felfelé haladunk a hierarchiában, az egység megváltoztatása hatással lehet a felette levő egységekre.

(Inkább a funkcionális, mint a strukturális tesztelés irányába mutat, de jó stratégia, különösen, ha objektumokat és egység újrafelhasználást tekintünk)



3. ábra Bottom-up tesztelés

4.1 JUnit tesztek

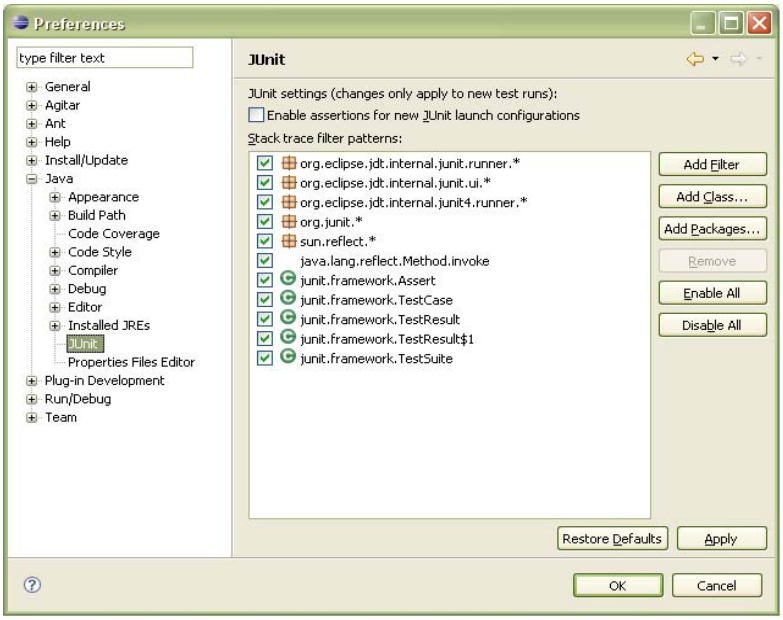
A JUnit eredetileg az IBM által fejlesztett, de jelenleg szabad forrású szoftverként hozzáférhető egységteszt-keretrendszer, melyet Erich Gamma es Kent Beck írták.

Fő funkciói a következők:

- tesztek automatikus futtatása egyben vagy részenként
- teszteredmény-áttekintés és kijelzés
- hierarchikus tesztstruktúra-támogatás
- tesztvégrehajtás többféle felületről

Néhány dolog, amire a JUnit nem képes:

- tesztek tervezése
- automatikus tesztprogram generálás
- lefedettség és teljesítménymérés



4. ábra JUnit beállításai

Az Eclipse tartalmazza a JUnit-ot. Grafikus felhasználói interfészt is kínál a tesztek futtatásához. Beállítása a *Preferences* ablakba történik. Általában az alapbeállításokon kívül mást nem kell beállítani. Meg lehet neki adni szűrőket, valamint beállíthatjuk, hogy milyen csomagok és osztályok jelenjenek meg a stack trace-ben.

Mit teszteljünk? Csak az API részek tesztelhetők, nincs lehetőség a GUI tesztelésére. Annak, amit tesztelni szeretnénk, láthatónak kell lennie.

4.1.1 Tesztesetek írása JUnit-hoz

A JUnit tesztesetekben gondolkodik, de az ő testcase fogalma egy kicsit eltér a szokásostól. Egy JUnit teszteset a tesztelendő komponens vagy komponensek bizonyos konfigurációját jelenti, amit felállítanak tesztelésre. Ebben a konfigurációban egy vagy több tesztfüggvény fut le, mind a teszteseten belül. A továbbiakban a JUnit teszteset fogalmát használjuk.

Egy JUnit teszteset egy Jáva osztály, amely a következőket tudja:

- a *TestCase* osztályból származik, ezzel jelzi a keretrendszernek, hogy futtatni kell. (Ez Junit 4-től már nem szükséges)

- felüldefiniálhatja a setUp() és tearDown() metódusokat, ezek az egyes tesztek kezdőállapotának inicializálásra és utána a takarításra szolgálnak (JUnit 4-ben @Before / @BeforeClass, valamint @After / @AfterClass annotáció)
- minden *test*-el kezdődő metódus tesztként lesz kezelve az osztályban
- a TestCase osztály *assert* kezdetű metódusaival tudja definiálni az elvárt működést, például az *assertEquals* azt ellenőrzi, hogy a két paramétere, az elvárt és a teszt során kapott objektum, egyenlő-e
- a testcase-eket tesztkészletekbe (testsuite) gyűjthetjük össze, így egyszerre tudjuk őket futtatni
- a tesztek futtatására grafikus és parancssori felületet is biztosít a keretrendszer, melyen megjelenik, hogy a futtatott tesztek közül melyiknek mi lett az eredménye

Azokat a beépített függvényeket, amelyeket a teszteléshez használunk, a junit.framework.Assert osztályból vesszük. Ezen assertXXX() függvények működtetik a tesztmetódusokat.
A legfontosabb *assert* metódusok:

- *assertEquals()*: két értéket hasonlítunk össze vele. A teszt sikeres, ha az értékek egyenlők
- *assertFalse()*: egy logikai kifejezést értékel ki. A teszt sikeres, ha a kifejezés hamis
- *assertNotNull()*: az objektum referenciát null-hoz hasonlítja. A teszt sikeres, ha a referencia nem null
- *assertNotSame()*: két objektum referencia memória címét hasonlítja össze az „==” operátor segítségével. A teszt sikeres, ha azok különböző objektumokra hivatkoznak
- *assertNull()*: egy objektum referenciát hasonlítja null-hoz. A teszt sikeres, ha az objektum referencia null
- *assertSame()*: ez a metódus szolgál a referencia egyenlőségre az „==” operátort használva. A teszt sikeres, ha ugyan arra az objektumra hivatkoznak
- *assertTrue()*: egy logikai kifejezést értékel ki. A teszt sikeres, ha a kifejezés igaz

Egy testcase elkészítéséhez létrehozunk egy új csomagot, a függőségekhez hozzáadjuk a *junit.jar* fájlt. Kiválasztjuk, hogy milyen csomagba szeretnénk tenni a tesztet, a *New* menüből, pedig kiválasztjuk a *Junit TestCase*-t, nevet adunk neki és létrejön az új testcase, melybe már írhatjuk is a tesztjeinket.

Egy teszteset vagy teszt futtatása: válasszunk ki egy JUnit osztályt vagy metódust, kattintsunk az egér jobb gombjával annak nevére, majd válasszuk ki a Run as -> JUnit test funkciót. A kiválasztott teszt vagy teszteset futása után az eredményt a JUnit view-ba látjuk.
Az ablakban szereplő információk:

- Piros/zöld a teszt eredménye hiba/sikeres
- Látható a meghiúsult tesztek neve
- Látható a hiba trace
- Látható a lefutott tesztek száma
- Látható a hibák száma

A JUnit tehát egy könnyen használható, rugalmas keretrendszer modultesztek futtatásához.
Egy egyszerű példán keresztül is bemutatom a működését. A tesztelendő metódus neve *add*, paraméterei két *int* típusú szám. A metódus a paraméterek összegét számolja ki.

```
public int add(int arg1, int arg2) {
    int result = 0;
    result = arg1 + arg2;
    int myInt = 0;
    return result + myInt;
}
```

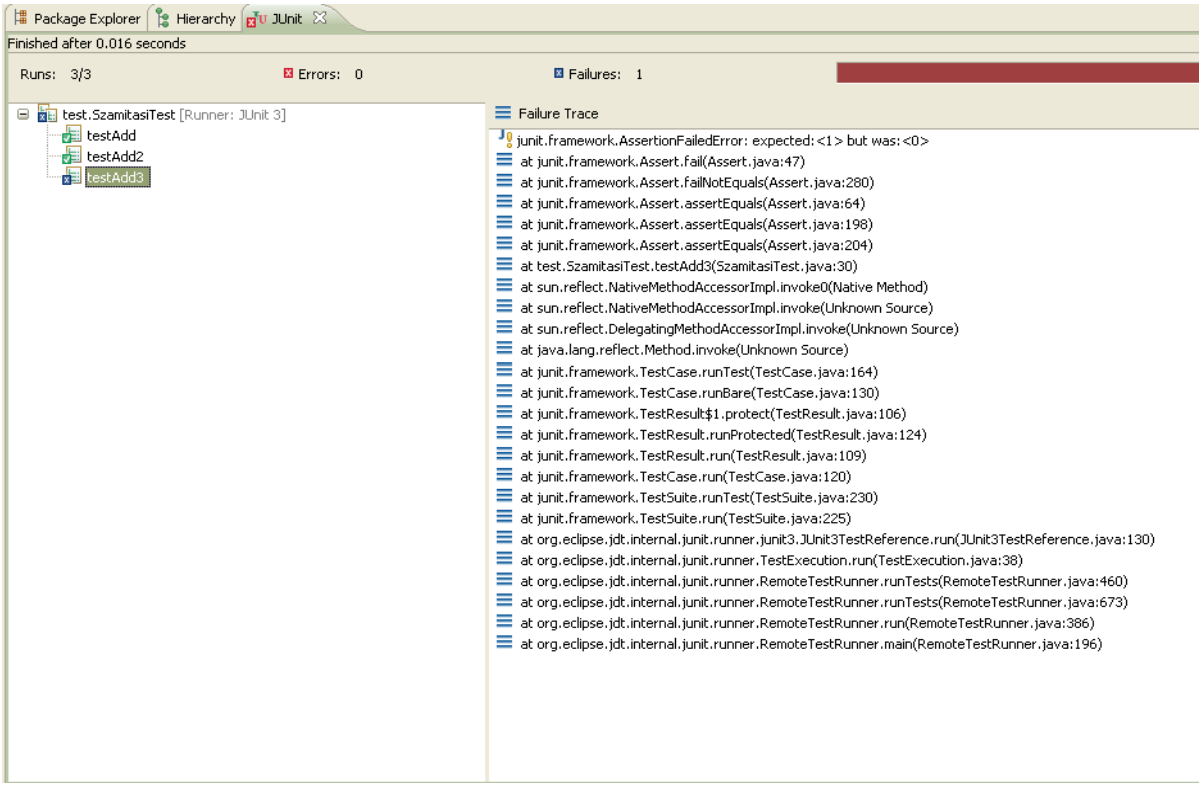
5. ábra Tesztelendő metódus

A Junit tesztesetbe három tesztelő metódust írtam a függvényre. A teszteset futása sikertelen lesz, mivel a harmadik teszt nem hibával tér vissza.

```
package test;

import src.Szamitasi;
import junit.framework.TestCase;

public class SzamitasiTest extends TestCase {

    Szamitasi szamitasi = new Szamitasi();

    public SzamitasiTest(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        super.setUp();
    }

    public void testAdd() {
        int addResult = szamitasi.add(1,1);
        assertEquals(2, addResult);
    }

    public void testAdd2(){
        int addResult = szamitasi.add(-1,1);
        assertEquals(0, addResult);
    }

    public void testAdd3(){
        int addResult = szamitasi.add(-1,1);
        assertEquals(1, addResult);
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }
```

6. ábra A tesztelendő metódus testcase-e

A tesztesetet lefuttatva a JUnit view ablak a következő információkat tartalmazza:



7. ábra JUnit view a testcase futása után

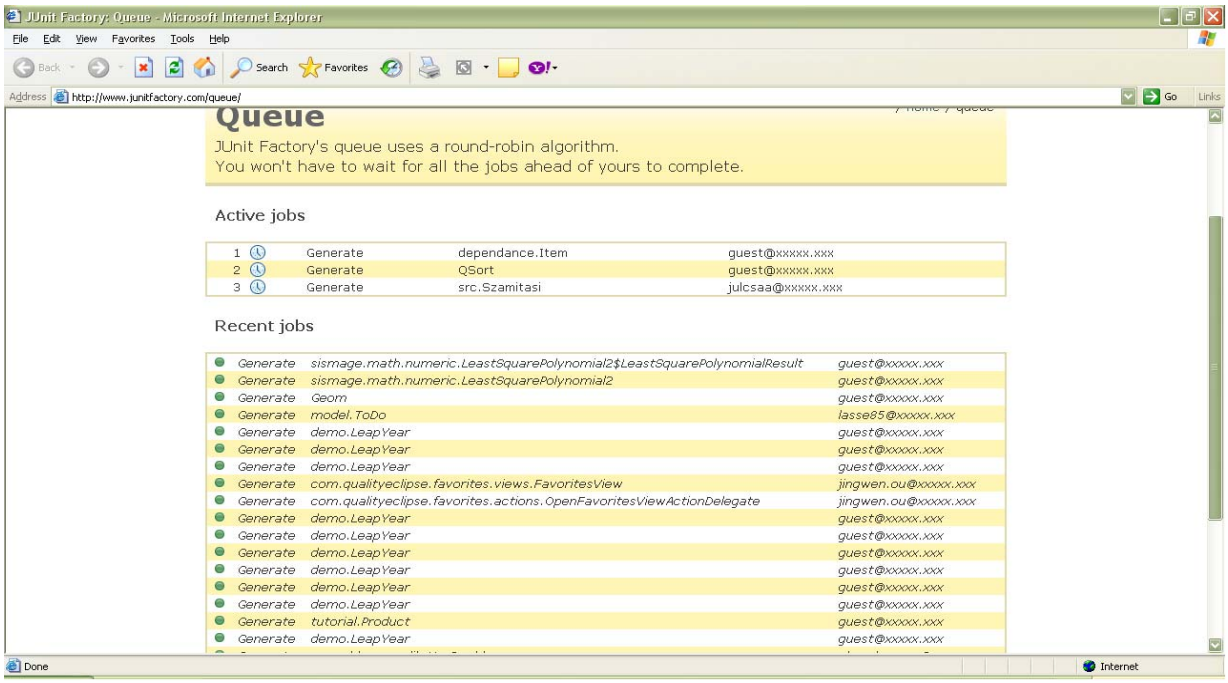
Jól látható, hogy három metódus futott le. Az Error-ok száma 0, de van egy Failure. A Failure az a hiba, amire számítunk. Ezekre fel vagyunk készülve. Az ok, amiért sikertelen volt a teszteset az ábrán is látszik, az elvárt eredmény 1 volt, míg a kapott eredmény 0 lett.

4.1.2 JUnitteszt generáló eszközök

Léteznek olyan eszközök melyek segítségével JUnit teszteket, teszteseteket tudunk generálni. Ezek között vannak fizetős és ingyenes eszközök. Egy ilyen ingyenes eszköz, melyet ki is próbáltam az a JUnit Factory, az Agitar Software terméke. A megírt Java kódra a JUnit Factory létrehoz egy JUnit testcase-t.

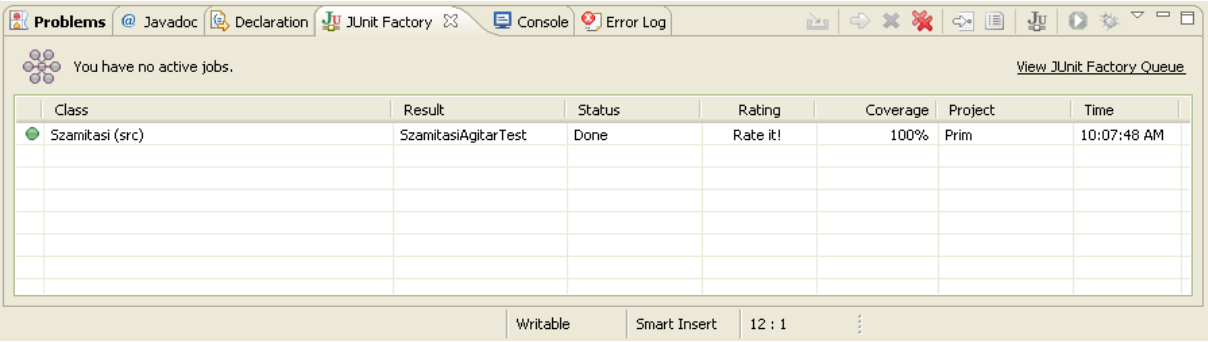
Telepítés után az Eclipse menüjébe bekerül az Agitar nevű új menü gomb. Kiválasztva egy megírt projektet, csomagot vagy osztályt és megnyomva a Generate Test gombot a kiválasztott adatok másolata el lesz küldve a JUnit Factory szervereinek. Rövid időn belül (pár másodperc vagy perc a projekt méretétől függően) megkapjuk a kiválasztott osztályokhoz generált teszteket.

Amíg várjuk az eredményt, addig egy ablakba figyelhetjük a változásokat, a generálás végén, pedig fontosabb információkat olvashatunk ki belőle. Az ablak hét oszlopból áll, melyek a következő adatokat tartalmazzák: a tesztelt osztály, a hozzá tartozó generált teszt, a generált teszt státuszát, értékelését, lefedettségét, a projekt nevét és azt az időt, amikor a generálás elkezdődött. A teszteket mi magunk is módosíthatjuk. Észrevettem, hogy alkalmanként ugyan arra az osztályra különböző teszteget generál az eszköz. Ennek oka, hogy Junit Factory különböző tesztgeneráló algoritmusokat tanulmányoz. A teszteket ezután *Run As->Agitar JUnit Test* funkcióval tudjuk futtatni. A futás eredményét a Junit ablakba kapjuk meg. Itt ellenőrizhetjük a teszt sikerességét.
A teszt és a tesztelendő osztályokat megnyitva láthatjuk melyek azok az utasítások, amelyek lefutottak és melyek azok, amelyek nem. Ez a kódfedettség szempontjából fontos.
Lássuk, az előbbi *add* függvényre milyen testcase-t generál az eszköz. A generálás elindítása után amíg várakozunk, a Junit Facotry view-ban a *View Junit Factory Queue*-ra kattintva a Junit factory oldalán megnézhetjük, hogy éppen hány job várakozik a queue-ban. Round-robin algoritmus szerint dől el, hogy a queue-ban várakozó job-ok közül melyik kerül kiszolgálásra.



8. ábra Junit Factory queue-ban várakozó job-ok

A view-t megnézve láthatjuk, hogy a generálás után a testcase le is futott. A kódfedése 100%.



9. ábra Junit Factory view

Most lássuk a SzamitasiAgitarTest osztályt, amit az eszköz generált. (A sorok elején szereplő zöld vonalak a kódfedést mutatják. Erről is írok majd a továbbiakban.) A generált testcase alig tér el az általam írott testcase-től.

```
 * Generated by Agitar build: JUnitFactory Version 2.2.0.000710 (Build

package src;

import com.agitar.lib.junit.AgitarTestCase;

public class SzamitasiAgitarTest extends AgitarTestCase {

    public Class getTargetClass()  {
        return Szamitasi.class;
    }

    public void testConstructor() throws Throwable {
        new Szamitasi();
        assertTrue("Test call resulted in expected outcome", true);
    }

    public void testAdd() throws Throwable {
        int result = new Szamitasi().add(100, 1000);
        assertEquals("result", 1100, result);
    }

    public void testAdd1() throws Throwable {
        int result = new Szamitasi().add(0, 0);
        assertEquals("result", 0, result);
    }
}
```

10. ábra A JUnit Factory által generált testcase

4.2 Kódfedés mérése

A modulteszt sikerességének mérőszáma a lefedettség (coverage). Ez azt jelenti, a tesztelendő modul hány százalékát "mozgatta meg" a testcase, vannak-e olyan részek, amelyek nem hajtódtak végre (tehát a hibáik nem derülhettek ki). Kétféle lefedettséget szokás használni modulteszt során: sor vagy függvénylefedettséget. Értelemszerűen a sorlefedettség azt mondja meg, a modul összes sorának hány százaléka hajtódott végre a teszt során, míg a függvénylefedettségnél a végrehajtott függvények/összes függvények arány számít. A JUnit nem képes önmagában lefedettséget mérni, erre más eszközök vannak, mint például a Rational PureCoverage, a Sitraka JProbe Coverage, a Clover vagy az EclEmma, viszont a keretrendszer kis mérete miatt kellemesen használható lefedettség mérő eszközökkel.

Fedési kritérium (coverage criterion): fontos, hogy egy tesztkészlet jóságát meg tudjuk határozni számszerűen, össze tudjuk hasonlítani egy másikkal. Ehhez nyújtanak segítséget a különböző fedettségi kritériumok. Ezek meghatározzák, hogy a tesztkészletnek bizonyos *tesztelési követelményeket* (*test requirement*s) teljesíteni kell, például a tesztesetek lefuttatása során a forráskód adott sorát érinteni kell. Így tudunk egy mérőszámot rendelni a tesztkészlethez, hogy a fedettségi kritérium által meghatározott követelmények hány százalékát teljesíti (például, ha a forrássoroknak csak a 80 százalékát hajtatja végre, akkor az összes forrássor lefedése kritériumot 80%-ban fedi le a tesztkészlet).

4.2.1 Kódfedési metrikák

Kódfedés mérésére többféle metrikát alkalmaznak. A legegyszerűbb az *utasításlefedettség (statement coverage)*, azaz, hogy a kódban szereplő utasítások hány százalékát hajtottuk legalább egyszer végre (szoktak kódsor lefedettséggel is hivatkozni erre a mérőszámra, de ez a kifejezés nem pontos azon nyelvek esetén, ahol egy sorban több utasítás is lehet). Ennél bonyolultabb mérőszám a *döntési ág fedettség (branch coverage)*. Ilyenkor azt mérjük, hogy a kódban szereplő

feltételek lehetséges kimenetei közül hányat fedtünk le. Száz százalékos döntési ág lefedettség esetén is kimaradhatnak fontos részek.

4.2.2 Kódfedési eszközök

Több kódfedési eszközzel is találkoztam, de ezek legnagyobb része fizetős eszköz, ezeket fogom bemutatni röviden a következő néhány sorban.

A *Pure Coverage* egy olyan kódfedési eszköz, melynek segítségével utasítás és metódus lefedettséget lehet mérni JAVA és C++ forráskódú programokhoz. Van grafikus felülete, ahol kiválaszthatjuk a futtatandó programot és az esetleges argumentumokat. A program futásának végégig gyűjti a fedési mérőszámokat. Előnye, hogy használata egyszerű. Hátránya, hogy csak a legalapvetőbb mérőszámokat ismeri.

A *Sitraka JProbe* egy integrált, több mindenre kiterjedő eszközrendszer melynek feladata, hogy megtalálja és kiküszöbölje a JAVA alkalmazásbeli kódhibákat és az eredménytelenséget. Négy különböző eszközzel segíti a fejlesztőket: JProbe Profiler, JProbe Memory Debugger, JProbe Threadalyzer es JProbe Coverage. A JProbe Coverage feladata a kódfedés mérése. A lefuttatott tesztek pontosak és minden részletre kiterjednek. Az eredmények megjeleníthetők html és szöveges formátumban is.

A *Clover* eszköz az Atlassian terméke, az Eclipse kiegészítője. Ezzel az eszközzel a fejlesztői környezetben tudjuk mérni a kódfedést. A Coverage Explorer ablakban jelennek meg az eredmények. A benne szereplő fa megmutatja a projekt és a benne szereplő csomagok, osztályok lefedettségét. A Clover menüje kényelmes, használata egyszerű. A tesztek futtatása után a Test Run Explorer nézettel jeleníthetjük meg az eredményeket, a státusz oszlopba látható hogy a teszt sikeresen lefutott vagy megbukott.

Az eddigi kódfedést mérő eszközökért fizetni kell. Ezeknek csak a próba verzióit tudtam letölteni, de egyes felsorolt eszközök próbaverziós működése nem teljes (például a *Clover* eszköz). Lássunk most egy olyan eszközt mely nyílt forráskódú.

Ez a kódfedési eszköz az *EclEmma*, mely egy ingyenes Eclipse kiegészítő. Használata nagyon egyszerű. Egér jobb gombjával kiválasztjuk, a megirt testcase-t, a menüből, pedig a *Coverage As → Junit Test*, ezzel a funkció elindul. Az eredményt a *Coverage view*-ba tekinthetjük meg, mely

a funkció indítása után automatikusan megjelenik. Itt olvashatjuk le a kódfedés mértékét. Eredményt akkor is látunk, ha a futtatott testcase-t es a benne hívott metódusok osztályát megnyitjuk. Az eredményt az utasítások színeiből látjuk.
Egy utasítás (sor) lehetséges három színe:

- piros, ha a sor egyáltalán nincs lefedve
- zöld, ha a sor teljesen le van fedve
- sárga, ha a sor csak részben van lefedve

A *Coverage view* a következő oszlopokat tartalmazza:

- *Element* fa hierarchia szerepel itt, ahol a fa gyökere maga a projekt, míg a levelek a vizsgált osztályok metódusai
- *Coverage* ez mutatja, hogy hány százalékos lett a fa adott elemeinek a kódfedése
- *Covered Instructions* a lefedett utasítások száma
- *Total Instructions* az összes utasítás száma

A Junit-hoz használt példa osztályt kibővítettem egy *coverageExample()* metódussal, melynek segítségével azt szeretném megmutatni, hogy bár a kódfedés mértéke 100%, még sincs minden lehetséges módon megmozgatva a metódus. A függvénynek három paramétere van. Ha a paraméterek értéke 1 vagy 2, akkor ennek megfelelően kap értéket három a metódusban deklarált változók. A változók: aa, bb, cc. A függvény egy *int* típusú számot ad vissza, amely a három változó aritmetikai műveletének eredmény: *aa/(bb-cc)*. A függvény érdekessége a nullával való osztás. A függvény tesztjébe beleírva ezt az esetet és futtatva a JUnit-ot *ArithmeticException*-t kapunk a nullával való osztás miatt. A teszt nem fut le. Megjegyzésbe téve azt a sort, ahol ezt az esetet tesztelem, a JUnit sikeresen lefut, a kódfedés pedig 100%-os lesz. A 100%-os kódfedés, mint már írtam azt jelenti, hogy a tesztelt metódus teljes egészében meg lett mozgatva és fény derült a lehetséges hibákra, egy egyszerű példával sikerült ezt megcáfolni.

Most lássuk a példákat. A tesztelendő osztály:

```
1   package src;
2
3   public class Szamitasi {
4
5       public int add(int arg1, int arg2) {
6           int result = 0;
7           result = arg1 + arg2;
8           int myInt = 0;
9           return result + myInt;
10      }
11
12      public int coverageExample(int a, int b, int c) {
13          int aa=0;
14          int bb=0;
15          int cc=0;
16
17          switch(a)
18          {
19              case 1: aa=10;break;
20              case 2: aa=20;break;
21          }
22          switch(b)
23          {
24              case 1: bb=1;break;
25              case 2: bb=2;break;
26          }
27          switch(c)
28          {
29              case 1: cc=1;break;
30              case 2: cc=3;break;
31          }
32
33          return aa / (bb-cc);
34      }
35  }
```

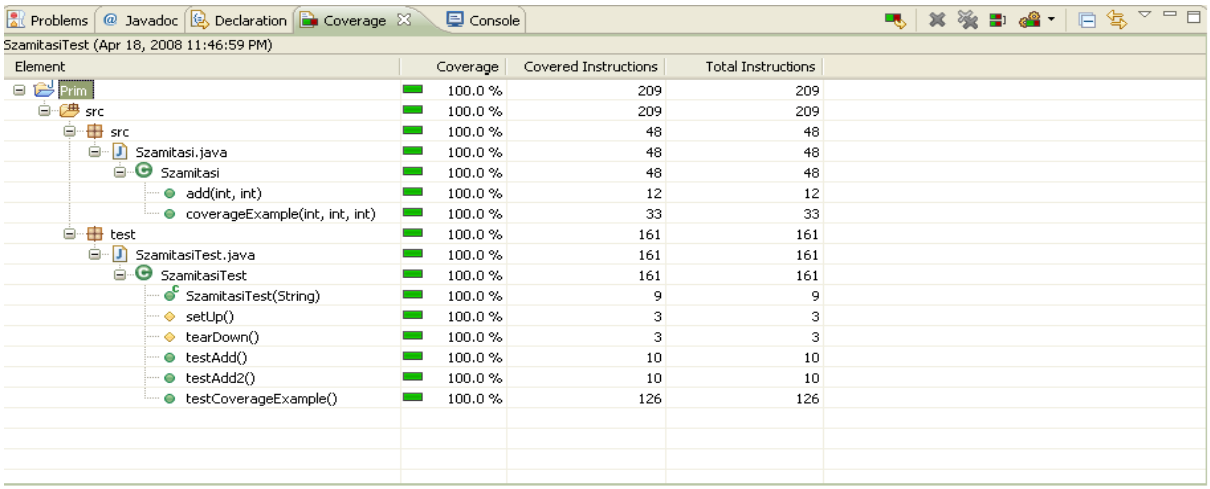
11. ábra A kódfedéshez használt osztály

Az osztály két függvényéhez tartozó tesztek:

```
13    protected void setUp() throws Exception {
14        super.setUp();
15    }
16    public void testAdd() {
17        int addResult = szamitasi.add(1,1);
18        assertEquals(2, addResult);
19    }
20
21    public void testAdd2(){
22        int addResult = szamitasi.add(-1,1);
23        assertEquals(0, addResult);
24    }
25
26    public void testCoverageExample()    {
27        int result1=0, result2=0, result3=0, result4=0, result5=0, result6=0;
28        result1 = szamitasi.coverageExample(1,2,1);
29        assertEquals(10, result1);
30        result2 = szamitasi.coverageExample(1,1,2);
31        assertEquals(-5, result2);
32        //assertEquals(0.0,szamitasi.coverageExample(2,1,1));
33        result3 = szamitasi.coverageExample(1,2,2);
34        assertEquals(-10, result3);
35        result4 = szamitasi.coverageExample(2,2,1);
36        assertEquals(20, result4);
37        result5 = szamitasi.coverageExample(2,1,2);
38        assertEquals(-10, result5);
39        result6 = szamitasi.coverageExample(2,2,2);
40        assertEquals(-20, result6);
41        System.out.print(result1 + ", " );
42        System.out.print(result2 + ", " );
43        System.out.print(result3 + ", " );
44        System.out.print(result4 + ", " );
45        System.out.print(result5 + ", " );
46        System.out.print(result6);
47    }
```

12. ábra A kódfedéshez használt JUnit osztály

A *Coverage view* tartalma:



13. ábra Coverage view

A következtetés az, hogy a 100%-os kódfedés sem garantálja azt, hogy a JUnit tesztjeinkkel minden lehetséges esetet levizsgáltunk és hibák már nem jöhetnek elő.

4.3 Tesztesetek

A megírt JUnit tesztosztályokat dokumentálni kell. Ezeket excel fájlokba fogjuk össze. Egy excel munkalap egy tesztnek felel meg, mely tartalmazza a teszt metódus bemeneti adatait és az elvárt eredményt.

A munkalap a következő adatokat kell, tartalmazza:

- *tesztazonosító*: a fejlesztési folyamat teljes menete során ezzel az azonosítóval hivatkozunk a tesztre. Hibajelentések készítésekor, illetve teszt riportok összeállításakor látjuk hasznát. Regressziós tesztelésnél különösen hasznos. Az én tesztazonosítóim *JTCx.y* formátumúak.
- *teszt*: ide kell írni a rövid leírását a tesztnek

- *készítette*
- *készítés dátuma*
- *végrehajtotta*
- *végrehajtás dátuma*
- *lezárás dátuma*
- *megjegyzés*: itt bővebben írhatunk a tesztről
- *lépés*: ez egy szám
- *teszt lépés*: a teszt lépései
- *végrehajtás és tesztadatok*: a tesztlépést hogyan hajtjuk végre, milyen bemeneti adatokat használunk
- *elvárt eredmény*: az elvárásunk, ami alapján el tudjuk dönteni, hogy jó-e a működés
- *eredmény*: a tényleges eredmény

JTC1.0				
Teszt:	**switch utasítás tesztelő**			
Készítette:	**Bodogan Judit**	Dátum: 2008-03-33		
Végrehajtotta:	Bodogan Judit	Dátum: 2008-03-33	Lezárva (dátum):	
Megjegyzés:				

Lépés	Teszt lépés	Végrehajtás és teszt adatok	Elvárt eredmény	Eredmény
1	Deklaráljuk a result1, result2, result3, result4, result5, result6 változókat	A változók kezdeti értéke 0		
2	A result1 változónak új értéket adunk	Meghívjuk a coverageExample függvényt 1,2,1 paraméterekkel. A függvény visszatérési értéke lesz a result1 új értéke	A result1 változó értéke 10	result1 értéke 10
		Meghívjuk az assertEquals metódust: az elvárt eredmény 10, a kapott eredmény a result1 értéke	A függvény True-val tér vissza	assertEquals True-val tér vissza
3	A result2 változónak új értéket adunk	Meghívjuk a coverageExample függvényt 1,1,2 paraméterekkel. A függvény visszatérési értéke lesz a result1 új értéke	A result2 változó értéke -5	result2 értéke -5
		Meghívjuk az assertEquals metódust: az elvárt eredmény -5, a kapott eredmény a result2 értéke	A függvény True-val tér vissza	assertEquals True-val tér vissza

4	A result3 változónak új értéket adunk	Meghívjuk a coverageExample függvényt 1,2,2 paraméterekkel. A függvény visszatérési értéke lesz a result3 új értéke	A result3 változó értéke -10 legyen	result3 értéke -10
		Meghívjuk az assertEquals metódust: az elvárt eredmény -10, a kapott eredmény a result1 értéke	A függvény True-val tér vissza	assertEquals True-val tér vissza
5	A result4 változónak új értéket adunk	Meghívjuk a coverageExample függvényt 2,2,1 paraméterekkel. A függvény visszatérési értéke lesz a result1 új értéke	A result4 változó értéke 20 legyen	result4 értéke 20
		Meghívjuk az assertEquals metódust: az elvárt eredmény 20, a kapott eredmény a result1 értéke	A függvény True-val tér vissza	assertEquals True-val tér vissza
6	A result5 változó új értéket adunk	Meghívjuk a coverageExample függvényt 2,1,2 paraméterekkel. A függvény visszatérési értéke lesz a result1 új értéke	A result5 változó értéke -10 legyen	result5 értéke -10
		Meghívjuk az assertEquals metódust: az elvárt eredmény 20, a kapott eredmény a result1 értéke	A függvény True-val tér vissza	assertEquals True-val tér vissza
7	A result6 változónak új értéket adunk	Meghívjuk a coverageExample függvényt 2,2,2 paraméterekkel. A függvény visszatérési értéke lesz a result1 új értéke	A result6 változó értéke -20 legyen	result6 értéke -20
		Meghívjuk az assertEquals metódust: az elvárt eredmény 20, a kapott eredmény a result1 értéke	A függvény True-val tér vissza	assertEquals True-val tér vissza
8	Az eredmény kiiratása	Kiiratjuk a képernyőre a valtozok értékeit	10, -5, -10, 20, -10, -20	10, -5, -10, 20, -10, -20

A teszteket teszt forgatókönyvekbe fogjuk össze.

Azonosító	Teszt	Teszt nap	Végrehajtás
JTC1.0	switch utasítás tesztelése	1	Y

A forgatókönyv részei:

- *azonosító*: a teszt azonosítója
- *teszt*: a teszt neve
- *teszt nap*: hány nap szükséges a teszt végrehajtásához

- *végrehajtás*: lehetséges értékei: Y-végrehajtva / N-nincs végrehajtva

A JUnit tesztekről készített dokumentumok fontosak. A tesztelést átláthatóbbá teszi, hiszen ha valami változik a forráskódban, akkor könnyen ki tudjuk keresni, hogy melyek azok a JUnit tesztek amelyeket újra futtatnunk kell.

4.4 Mock object

Egységtesztelés során egyszerre egy metódust tesztelünk. A tesztelés körülményessé válhat, ha ez a metódus több más objektumtól is függ, főleg ha ezek az objektumok erőforrás igényesek, mint például adatbázis vagy hálózati kapcsolat. Ha nem vigyázunk, majdnem a teljes alkalmazás környezetet el kell indítani ahhoz, hogy le tudjuk tesztelni a metódusunkat. Ez nemcsak költséges és időigényes, hanem túlságosan is hozzáköti a tesztünket a környezethez.
A problémára a Mock objektumok adják a választ. Egy mock objektum egy másik objektum interfészét utánzó, csak nyomkövetési és tesztelési célokra szánt objektum. Ha a tesztelendő metódusunkat ezekkel a mock objektumokkal kapcsoljuk össze, akkor izoláltan tesztelhetjük azt plusz erőforrások igénybevétele nélkül.

4.5 Integrációs tesztek

Az alkalmazásoknak vannak fejlesztési felelősei (rendszerszervezők), akik az alkalmazás önálló működését, illetve az új funkciókat le tudják tesztelni. Viszont az alkalmazások a legritkább esetben működnek önállóan. Fontos, hogy az alkalmazást a többi alkalmazással összekapcsolva, azokkal kommunikálva is leteszteljük. Ezek a tesztek derítik ki a legköltségesebb hibákat.
Az integrációs tesztek a modulok rendszerbeli együttműködését vizsgálja. A kooperáló modulokból felépített rendszerek állapotát az egyes modulok közötti interakciók is meghatározzák.
Ezt a tesztelést szokás még Grey-boksz tesztelésnek is hívni.

Az alrendszer teszt (subsystem test): egy alrendszer, azaz az alrendszerbe integrált modulok együttesének tesztelése. Az alrendszerben lévő modulok közti interface-hibák detektálása a fő cél. Az első alkalom, ahol már funkcionális kérdések is előkerülhetnek, ugyanis egy alrendszerhez, már funkció köthető. Itt már eldönthető, hogy a felhasználó által felmerülő kéréseket teljesiti-e a rendszer vagy sem.

A rendszer teszt (system test): az egyes alrendszerek közti kommunikációt, azaz a programstruktúrában egymáshoz kapcsolódó elemek együttműködését, valamint az alrendszer rendszerbe való integrálását, kapcsolódását vizsgálja, hogy az megfelelő-e. Az együttműködési hibáknak sok oka lehet, pl információk, adatok elveszhetnek, vezérlőjelek elveszhetnek.

Elképzelhető, hogy egy rendszer annak ellenére hibásan működik, hogy az összes alkotó komponense egyenként korrekt.

- *Bing-bang tesztelés:* az összes modult egyszerre rakjuk össze, a teljes komplexumra hajtjuk végre a tesztelést.
- *Inkrementális tesztelés:* egyenként tesztelünk, majd bővítünk. Minden bővítés után az összeállt komplexumot leteszteljük.

Fő nehézsége ennek a tesztelésnek a hiba lokalizálása, éppen ezért inkább az *Inkremtális tesztelés* javasolt.

4.6 Tesztkörnyezet

Nagyon fontos szempont, hogy a rendszer fejlesztése során folyamatosan ellenőrizni lehessen, hogy az eddig elkészült részek megfelelnek-e az elvárásoknak, tartalmaznak-e hibákat. Ezért ajánlatos egy tesztkörnyezetet létrehozni, ahol a rendszer működéséhez szükséges konfigurációkat be kell állítanunk. Így ellenőrizni tudjuk az eredményeket, illetve ha a program mégsem működik jól, akkor a hiba helyéről és okáról is kaphatunk információt.

Amire szükség lehet egy tesztkörnyezet létrehozásában: egy webszerver, egy adatbázis szerver, ez lesz a tesztszerver és fejlesztői környezet. Fontos hogy jól állítsuk be a szervereket.

A rendszer működéséhez, teszteléséhez szükséges törzsadatokat fel kell venni a tesztadatbázisba. A törzsadatokon kívül, ha szükséges létrehozhatók benne csomagok, függvények, eljárások,

szekvenciák. A törzsadatokhoz feltöltő scriptek-et kell készíteni. Legjobb őket egy-egy excel munkalapon megírni, mert ha új mező kerül az adatbázis sémába, akkor gyorsabb a script modosítása. Íme a példa rá:

ID	FIRSTNAME	LASTNAME	ADDRESS	CUSTOMER	Column	Value
1	Nagy	Anna	4029. Db. Csapó u. 75.	INSERT INTO CUSTOMER (FIRSTNAME,LASTNAME,ADDRESS) VALUES ('Nagy','Anna','4029. Db. Csapó u. 75.');	FIRSTNAME,LASTNAME,ADDRESS	'Nagy','Anna','4029. Db. Csapó u. 75.'
2	Kiss	Andrea	4032. Db. Egyetem sgt. 49.	INSERT INTO CUSTOMER (FIRSTNAME,LASTNAME,ADDRESS) VALUES ('Kiss','Andrea','4032. Db. Egyetem sgt. 49.');		'Kiss','Andrea','4032. Db. Egyetem sgt. 49.'
3	Hajas	Imre	4032. Füredi út 25.	INSERT INTO CUSTOMER (FIRSTNAME,LASTNAME,ADDRESS) VALUES ('Hajas','Imre','4032. Füredi út 25.');		'Hajas','Imre','4032. Füredi út 25.'

Ezzel a Customer táblába tudok beszúrni három új sort. Az ID, FIRSTNAME, LASTNAME, ADDRESS oszlopnevek a tábla oszlopai. A cellák a táblázat mezőinek lehetséges értékeit tartalmazzák. A Column oszlopba a tábla oszlopainak nevét fűztem össze. A Value oszlopba az ID, FIRSTNAME, LASTNAME, ADDRESS oszlopok celláinak értékeit fűztem össze. Végezetül a CUSTOMER oszlopba összefűztem magát az insert utasítást a Column és Value oszlopok celláiban található értékek segítségével.

Az így elkészült három *insert* script:

INSERT INTO CUSTOMER (FIRSTNAME,LASTNAME,ADDRESS) VALUES ('Nagy','Anna','4029. Db. Csapó u. 75.');

INSERT INTO CUSTOMER (FIRSTNAME,LASTNAME,ADDRESS) VALUES ('Kiss','Andrea','4032. Db. Egyetem sgt. 49.');

INSERT INTO CUSTOMER (FIRSTNAME,LASTNAME,ADDRESS) VALUES ('Hajas','Imre','4032. Füredi út 25.');

A scripteket a fejlesztő csapat tesztelői szokták elkészíteni.

A scriptekről release készül. A fejlesztés során több script release is fog készülni, mivel a fejlesztés során gyakran változik a modell. Az első sql release-en kívül az összes többi csak az adatbázis séma változásait, vagy plusz törzsadatokat tartalmazza: új sorok beszúrása valamely törzsadat táblába, új mező beszúrása a séma valamely táblájába, már meglévő sorok módosítása táblázatokban, táblázat mezőjének törlése, rekord törlése táblázatokból, …. Minden release-ben össze kell fogi a futtatandó script-eket egyetlen script-be, azért, hogy egyszerűbb legyen az adatbázis létrehozása.

A tesztelés során többször is szükség van a teszt adatbázis ismételt létrehozására. A Selenium IDE által rögzített tesztesetek futtatásakor van ennek lényeges szerepe. (Vegyük csak az én példaalkalmazásomat, melyet a 6. Fejezetben fogok leírni, ahányszor lefuttatom az új ügyfél hozzáadás esetet, az új ügyfél annyiszor kerül be az adatbázisba különböző azonosítókkal.)

Az sql script-ek karbantartása egy nagyobb rendszer fejlesztésénél nagy munka. Általában egy tesztelő tartja ezt karban, ő vizsgálja a megírt script-ek helyességét a tesztadatbázison. A tesztelő jó ha odafigyel arra is, hogy a külső kulcsok jó rekordra mutatnak-e. Ha valami nem működik, akkor értesítenie kell a script készítőjét a hibabejelentő rendszeren keresztül.

5 Rendszertesztelés

Ez a tesztelési fázis a megfogalmazott céloknak a követelmények meghatározásakor specifikációvá történő alakításánál elkövetett hibák feltárását szolgálja.

- *Szolgáltatás tesztelés*: a rendszer nyújtotta követelmények, nem a rendszer specifikációra épül
- *Mennyiségi tesztelés*: nagy mennyiségű adatokkal való tesztelés, kapacitás korlátok ellenőrzésére. Viszonylag erőforrás igényes
- *Stressz terheléses tesztelés*: nagy mennyiségű adat rövid idő alatti feldolgozása. A rendszer valós helyzetekbeli robosztusságának ellenőrzésére végeznek olyan *stressztesztek*, melyek a valóságban elő nem forduló feltételeket jelentenek

- *Használhatósági tesztelés*: a használat közbeni minőség azt jelenti, hogy egy termék egy meghatározott felhasználó által, egy meghatározott felhasználási körben használva, meghatározott célok hatékony és produktív elérésére, mennyire kielégítő és mennyire vezet megelégedéshez
- *Konfigurációs tesztelés*: a program helyesen működik minden olyan számítógép konfiguráció és képernyőfelbontás mellett, amelyet a Kereskedelmi csoport követelményként meghatároz. A *konfigurációs tesztelés* sok esetben, különböző konfigurációkban elvégzendő teljesítménytesztelést jelent
- *Erőforrás tesztelés*: a program helyesen működik csökkentett memória és merevlemez-terület mellett is
- *Felhasználói dokumentáció és a program súgójának tesztje*: a program súgója és dokumentációja összhangban van a program valós működésével. A dokumentációk ellenőrzése nyelvi szempontból
- *Biztonsági tesztelés*: célja, hogy felfedje az adatvédelemmel és adatbiztonsággal kapcsolatos hibákat. A tesztesetek létrehozásához előnyös lehet a hasonló rendszerek már felderített biztonsági hiányosságainak ismerete
- *Teljesítmény tesztelés*: egyes rendszereknél a teljesítmény vagy a hatékonyság specifikálva van különböző terheléseknél és konfigurációkra meghatározott válaszidők és feldolgozási sebességek formájában
- *Regressziós teszt*: a program már megvizsgált és jónak talált részei új programrészek beépítése után is jók maradnak.

6 Funkcionális tesztelés

A funkcionális teszt (más néven validációs vagy érvényesítési teszt) feladata a használatieset-modellben és az egyéb funkcionális követelményekben rögzített folyamatok, forgatókönyvek végrehajthatóságának és helyességének ellenőrzése.

A funkcionális tesztben a teszteseteket így hosszabb felhasználói interakciók forgatókönyvei alkotják. Nem célja a funkcionális teszteknek a forráskódbeli metódusok egyenkénti ellenőrzése, az üzemeltetési funkciók (telepítés, migrálás) és a felületergonómiai követelmények értékelése.
A teszt célja a követelményekben rögzített felhasználói célok és interakciók, üzleti folyamatok ellenőrzése. Ez specifikáció alapján történő tesztelés, ezért szoktak néha így is hívni. Másik elnevezése a fekete-doboz (black-boksz) tesztelés, mégpedig azért mert a tesztek tervezésekor és végrehajtásakor a tesztelés alatt álló rendszer belsejébe nem látunk bele.
A példaalkalmazás:
Az alkalmazás, amelyen keresztül bemutatom a funkcionális tesztelést és a hozzá tartozó teszteseteket, az egy egyszerű ügyfélkezelő alkalmazás. Az alkalmazás főfunkciói:

- *Új ügyfél hozzáadása*
  Ügyfél személyes adatai (mindhárom mező kitöltése kötelező):
  - Vezetéknév
  - Keresztnév
  - Cím

  Ügyfél felvitele gombra kattintva rögzítjük az ügyfelet, és lehetőségünk van számlát rögzíteni az ügyfélhez
  Vissza linkre kattintva nem adunk meg számlát az ügyfélhez, hanem visszalépünk a főoldalra
- *Ügyfél számláinak megadása* (mindkét mező kitöltése kötelező):
  - Számlaszám
  - Egyenleg

  Hozzáadás funkció kattintva rögzítjük az ügyfél számláját, az új számla bekerül egy táblázatba. Akárhány számlát rögzíthetünk az ügyfélhez.
  Módosítás funkció: a számlák táblázatából a sorok elején szereplő rádió gombok segítségével valamelyik sort kiválasztva az adatok betöltődnek a megfelelő mezőkbe. Átírva a mezőket (valamely mezőt) és megnyomva a *Módosít* gombot, a számla adatai módosulnak
  Törlés funkció: a számlák táblázatából valamelyik sort kiválasztva az adatok betöltődnek a megfelelő mezőkbe. A *Törlés* gombra kattintva a sor törlődik a táblázatból

Vissza linkre kattintva az ügyfél felvitelre lépünk

Vissza a főoldalra linkre kattintva a főoldalra lépünk

- *Ügyfél keresés*
  - *Vezetéknév*
  - *Keresztnév*
  - *Cím*

  Kikereshetünk rögzített ügyfeleket vezetéknév, keresztév és cím alapján.

  Keresés: kilistázódnak a keresési feltételeknek megfelelő ügyfelek és adatik

  Kiválasztva valamelyik sort (bejelölve az elején szereplő rádió gombot) az ügyfél adatai betöltődnek a megfelelő mezőkbe.

  Számlák megtekintése funkció: Kiválasztva valamelyik sort a kilistázott ügyfelek közül (bejelölve az elején szereplő rádió gombot) az ügyfél adatai betöltődnek a megfelelő mezőkbe. A Számlák megtekintése gombra kattintva az Ügyfél számláihoz lépünk. A már rögzített számlák szerepelnek a képernyőn, de lehetőségünk van új számla rögzítésére is

  Módosítás funkció: Kiválasztva valamelyik sort a kilistázott ügyfelek közül (bejelölve az elején szereplő rádió gombot) az ügyfél adatai betöltődnek a megfelelő mezőkbe. Módosítva valamely mezőt és rákattintva a *Módosítás* gombra az ügyfél adatai módosulnak

  Törlés funkció: Kiválasztva valamelyik sort a kilistázott ügyfelek közül (bejelölve az elején szereplő rádió gombot) az ügyfél adatai betöltődnek a megfelelő mezőkbe. A *Törlés* gombra kattintva az ügyfél és számlái törlődnek

  Vissza linkre kattintva az ügyfél felvitelre lépünk

  Vissza a főoldalra linkre kattintva a főoldalra lépünk

6.1 Funkcionális tesztelési módszerek

Az ekvivalencia particionálás: abból indulunk ki, hogy minden hibához csak egy tesztesetet definiáljunk. Ennek érdekében a szoftver bemeneti adatainak minden kombinációját tartalmazó képzeletbeli halmazt olyan részekre (partíciókra) bontja, melyek mindegyike más és más hiba

felderítésére alkalmas. A bemenetek partícionálása után, a tesztesetek definiálásakor az egyes partíciókból egyet-egyet kell kiválasztani

A határérték analízis: az ekvivalencia partícionálás finomítása. Azt definiálja, hogy melyik bementet válasszuk az ekvivalencia osztályból. Azt mondja, hogy ha két partíció határos, vagyis a bemenetek természetes sorrendjében egymás után következő bemeneteket tartalmaz, akkor ezeket a partíció határán levő értékeket célszerű a tesztesetekben használt bemenetnek választani. Egy partícióból, ha több más partícióval határos, akár több bemenetet is használhatunk.

6.2 Selenium (Web tesztelés kliens oldali kódokkal)

A *Selenium* tesztelő eszköz arra vállalkozott, hogy böngésző automatizálással teljes értékű webteszteket hajtson végre. Az eszköz négy részből áll:

- Selenium Core
- Selenium IDE
- Selenium RC
- Selenium on Rails

A Seleniummal a tesztek a böngészőben futnak, mintha felhasználó futtatná őket. A tesztek megírása egyszerű. 100% Javascript és HTML kódok. Futtathatunk egész forgatókönyveket, vagy egyetlen tesztesetet. Funkcionális tesztelésre, regressziós tesztelésre és unittesztelésre is alkalmas. A tesztelés böngésző kompatibilis, vagyis az alkalmazást különböző böngészőkben és operációs rendszereken próbálhatjuk ki.

A *Selenium* által támogatott platformok:

- Windows:
    - Internet Explorer 6.0
    - Firefox 0.8 to 2.0
    - Mozilla Suite 1.6+, 1.7+
    - Seamonkey 1.0
    - Opera 8
- Linux:

- Firefox 0.8 to 2.0
- Mozilla Suite 1.6+, 1.7+
- Konqueror

A Test suite-ok (forgatókönyvek) teszteket (teszteseteket) tartalmaznak. (Teszteset: adott funkció adott paraméterekkel adott környezetben való lefuttatása, az elvárt eredmény megjelölésével. Teszt forgatókönyv: regressziós egység, amit a tesztesetek sorrendje határoz meg.) Az egymásra épülő teszteket forgatókönyvbe fogjuk össze. Ezek egyszerű táblázatok. Egyetlen kattintással több tesztet is le tudunk futtatni.

A Selenium Core-ban a TestSuite.html tartalmazza a teszteseteket. Ez egy olyan táblázat melynek egyetlen oszlopa van, itt vannak fel sorolva a tesztesetek.

Egy HTML táblázat három oszloppal:

- Első oszlop (*Command*): Selenium command
- Második oszlop (*Target*): első paraméter (kötelező megadni)
- Harmadik oszlop (*Value*): második paraméter (opcionális)

Fogalmak:

- Element Locators
- Patterns
- Action
- Accessors (letárolja az eredményeket)
- Assertion

Element Locators:

- id=id
- name=name
- identifier=id
- dom=javascriptExpression
  - Bármilyen – egy elem megkeresésére, hivatkozására használható – JavaScript kifejezés használható

    element = document.forms[‘myForm’].myDropDown

- xpath=*xpathExpression*
  - Beazonosít egy elemet a DOM fában egy XPath kifejezés alapján

    xpath=//table[@id='table1']//tr[4]/td[2]
- link=*textPattern*
  - A megadott mintára illeszkedő szöveget, tartalmazó linket választja ki

Patterns:

- glob:pattern
  - Vizsgálja a string illeszkedését a megadott mintára, melyben a '*' több karaktert jelent, a '?' pedig egy karaktert
- regexp:regexp
  - Reguláris kifejezés alapján való illeszkedést vizsgálja
- exact:string
  - Megadott stringre való pontos illeszkedést vizsgálja

Action:

- A felhasználó olyan tevékenységei, amivel az alkalmazásba action-ket váltunk ki (pl. kattintás egy gombra, írás beviteli mezőbe)
- action: végrehajtja a parancsot
- actionAndWait: végrehajtja a parancsot és megvárja a választ

Assertion:

- Az alkalmazás állapotát kérdezhetjük le
- Három formája van:
  - assert (hiba esetén leáll a futás)
  - verify (hiba esetén a futás nem áll le)
  - waitFor (feltétel + timeout)

6.2.1 Selenium IDE

A Firefox kiterjesztése. Integrált fejlesztői környezet. Segítségével teszteket vehetünk fel, játszhatunk vissza egyszerűen. Lépésenkénti visszajátszás: Debug, breakpoints. A tesztek lementhetőek több formátumba: HTML, JAVA, Ruby scripts, …. Van lehetőség saját formátum létrehozására is. Lássunk erre egy példát:

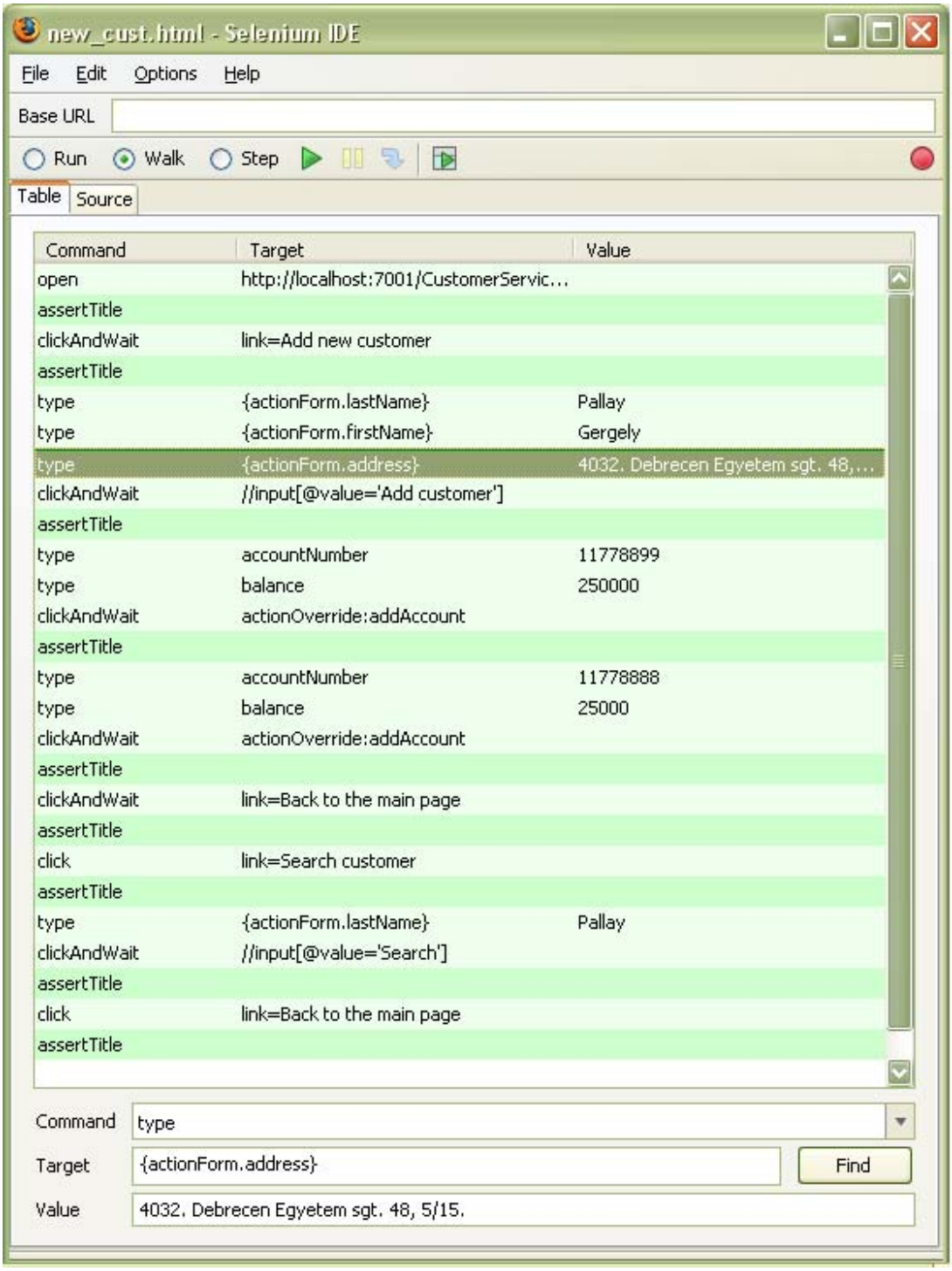
```
var SEPARATORS = {
    comma: ",",
    tab: "\t"
};

function formatCommands(commands) {
    var result = '';
    var sep = SEPARATORS[options['separator']];
    for (var i = 0; i < commands.length; i++) {
        var command = commands[i];
        if (command.type == 'command') {
            result += command.command + sep + command.target + sep + command.va
        }
    }
    return result;
}

function parse(testCase, source) {
    var doc = source;
    var commands = [];
    var sep = SEPARATORS[options['separator']];
    while (doc.length > 0) {
        var line = /(.*)(\r\n|[\r\n])?/.exec(doc);
        var array = line[1].split(sep);
        if (array.length >= 3) {
            var command = new Command();
            command.command = array[0];
            command.target = array[1];
            command.value = array[2];
            commands.push(command);
        }
        doc = doc.substr(line[0].length);
    }
    testCase.setCommands(commands);
}

function format(testCase, name) {
    return formatCommands(testCase.commands);
}

options = {separator: 'comma'};

configForm =
    '<description>Separator</description>' +
    '<menulist id="options_separator">' +
    '<menupopup>' +
    '<menuitem label="Comma" value="comma"/>' +
    '<menuitem label="Tab" value="tab"/>' +
    '</menupopup>' +
    '</menulist>';
```

14. ábra Saját formátum létrehozása Selenium IDE-ben

Intelligens html elem (gomb, input) azonosítás Id, name, XPath alapján
Működés:

- Elindítjuk a Selenium IDE funkciót
- Bekapcsoljuk a felvételrögzítést
- Minden művelet, amit a böngészőben végrehajtunk rögzítésre kerül

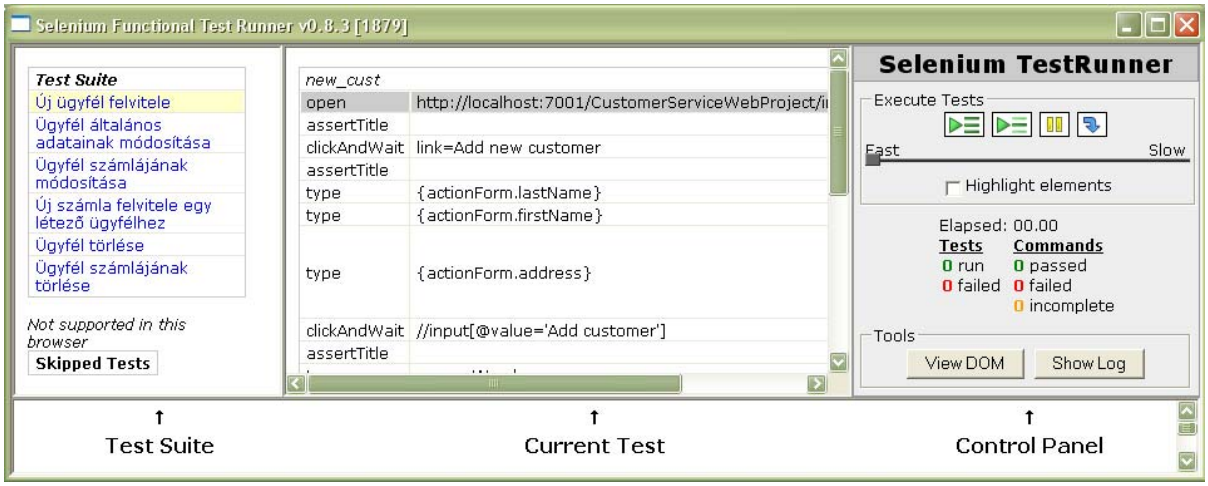
A rögzítés gomb kikapcsolása után lementhetjük a felvételt. A Play vagy Play with Selenium TestRunner gombokkal újra lejátszhatjuk azt. A Log ablakban figyelhetjük, hogy épp melyik sor hajtódik végre, illetve a rendszerüzeneteket. A teszteket mi magunk is szerkeszthetjük. Utasítás sorokat írhatunk a már kész tesztbe felhasználva a beépített függvényeket.
Van lehetőség a rögzített vagy megírt tesztek összefogására, egységes futtatására. Ez nagyon egyszerű: a Selenium core-ban található TestSuite.html fájl azon részét, ahol a tesztesetek szerepelnek átírjuk, beírjuk a mi tesztjeinket, majd futtatjuk a TestRunner.hta fájlt. Az ablakban megnyomva a Go gombot már a mi teszt eseteink töltődnek be.
Kipróbáltam a Selenium IDE-t az alkalmazásomon többek között úgy, hogy egy új ügyfél felvitelét rögzítettem vele. Az új ügyfélnek megadtam a nevét címét, aztán rögzítettem hozzá két számlát, majd ellenőrzésként megkerestem az ügyfelet a vezetékneve alapján.
Az ábrán jól látszanak a lépések, melyeket a böngészőben hajtottam végre. A zöld sorok azt jelzik, hogy a rögzített tesztesetet újra lejátszottam és sikeresen végrehajtódott annak minden sora.



15. ábra Új Ügyfél rögzítése a Selenium IDE-vel

Selenium Core

A rögzített teszteseteket html formátumba mentettem el. Ezeket bemásolva a *Selenium Core tests* könyvtárába, a *TestSuite* fáljába pedig átírva a táblázatok sorait, oda az én rögzített teszteseteimet beírva a *TestRunner*-ba már a saját teszteseteimet futtathatom. A *TestRunner* a következő képen néz ki:



16. ábra TestRunner.hta

6.3 Tesztesetek, tesztforgatókönyvek

Nagyon fontos, hogy az üzleti követelményrendszerből állnak elő a magas szintű tesztesetek (testcase).

A teszteset definíciói:

- Adott funkció adott paraméterekkel adott környezetben való lefuttatása, az elvárt eredmény megjelölésével
- Lépéseket tartalmazó véges sorozat, mely a rendszer egy futását határozza meg
- Egy adott hiba felderítésére szolgáló teszt

Egy teszteset lefutása után a környezetnek ugyanabba az állapotba kell kerülnie, mint amilyenben volt a futás előtt. Ez igaz, minden tesztesetnek ki kell takarítania maga előtt és maga után a saját közvetlen környezetét.
A jó teszteset az, ami nagy valószínűséggel egy még felderítetlen hibát mutat ki a programban.
Néhány dolog, amit jó szem előtt tartani a tesztesetek készítésekor:

- a meg nem ismételhető tesztesetek kerülni kell
- teszteseteket mind az érvénytelen, mind az érvényes adatokra kell készíteni.
- minden tesztesetből a lehető legtöbb információt ki kell hozni, azaz minden teszteset eredményét alaposan végig kell vizsgálni. Ezzel jelentősen csökkenthető a szükséges próbák száma
- egy próba eredményeinek vizsgálata során egyaránt fontos megállapítani, hogy miért nem valósít meg a program valamilyen funkciót, amit elvárunk tőle, illetve, hogy miért végez olyan tevékenységeket is, amelyek nem feltételeztünk róla
- a program tesztelését csak a program írójától különböző személy képes hatékonyan elvégezni

Forgatókönyv: Regressziós egység, melyet a tesztesetek sorrendje határoz meg.
Sajnos nem igazán elterjedt, de annál inkább javasolt a tesztesetek prioritásának meghatározása a teszt funkcionális terjedelmének kialakításával összhangban, gazdasági kihatás és használati gyakoriság alapján. Az állandó időnyomás miatt meg kell határozni azokat a teszteseteket, amelyeket feltétlenül végre akarunk hajtani. A teszteseteket excel táblázatokba rögzítjük.
Egy teszteset a következő adatokat kell tartalmazza:

- *teszteset azonosító*: a fejlesztési folyamat teljes menete során ezzel az azonosítóval hivatkozunk a tesztesetre. Hibajelentések készítésekor, illetve teszt riportok összeállításakor látjuk hasznát. Regressziós tesztelésnél különösen hasznos. Az én példámban az azonosítók *TC*-vel kezdődnek, a rövidítés a TestCase-ből ered
- *teszteset:* ide kell írni a teszteset nevét
- *készítette*: a tesztelő, aki a tesztesetet készítette
- *készítés dátuma*

- *tesztfeltételek*: a teszteset végrehajtásának előfeltételei (például milyen tesztesetet kell előtte hibamentesen végrehajtani)
- *végrehajtotta*: a tesztelő, aki végrehajtotta a tesztesetet
- *végrehajtás dátuma*
- *lezárás dátuma*
- *megjegyzés*: itt bővebben kifejthetjük a tesztesetet
- *lépés*: egy szám
- *tesztlépés*: a teszteset lépései
- *végrehajtás és tesztadatok*: a tesztlépest hogyan hajtjuk végre, milyen bementi adatokat használunk
- *elvárt eredmény*: az elvárásunk, ami alapján el tudjuk dönteni, hogy jó-e a működés
- *eredmény*: a tényleges eredmény. Ezt akkor fontos kitölteni, ha az elvárttól eltérő eredményt kapunk

A Selenium által rögzített teszteset dokumentálva a következő képen néz ki:

TC_CUST1.0				
Teszteset:	**Új ügyfél rögzítése**			
Készítette:	**Bodogan Judit**	Dátum: 2008-03-23		
Végrehajtotta:	Bodogan Judit	Dátum: 2008-03-23	Lezárva (dátum):	
Teszt feltételek:				
Megjegyzés:	Az ügyfélhez rögzítünk két számlát, ellenőrzésként rögzítés után megkeressük az ügyfelet.			

Lépés	Teszt lépés	Végrehajtás és teszt adatok	Elvárt eredmény	Eredmény
1.	Új ügyfél rögzítése	A kezdő képernyőn kattintsunk az "Új ügyfél hozzáadása" linkre	Az "Ügyfél általános adatai" képernyőre lépünk	
	Ügyfél általános adatai képernyő/ Ügyfél adatainak megadása:	Vezetéknév: Pallay		
		Keresztnév: Gergely		
		Cím: 4032. Debrecen Egyetem sgt. 48, 5/15.		

2	Ügyfél felvitele	Kattintsunk az "Ügyfél felvitele gombra"	Az "Ügyfél számlái" képernyőre lépünk	
	Ügyfél számlái képernyő/ Számla adatainak megadása	Számlaszám: 11778899		
		Egyenleg: 250000		
	Hozzáadás	Kattintsunk a "Hozzáadás" gombra	A számlák táblázatába bekerül az újonnan felvett számla azonosítója, száma és egyenlege	
3	Vegyünk fel egy újabb számlát az ügyfélhez:	Számlaszám: 11778888		
		Egyenleg: 25000		
	Hozzáadás	Kattintsunk a "Hozzáadás" gombra	A számlák táblázatába bekerül az újonnan felvett számla azonosítója, száma és egyenlege. Az ügyfélnek most már két számlája van	
	Vissza a főoldalra	Kattintsunk a "Vissza a főoldalra" linkre	A főoldalra lépünk	
4	Kezdő képernyő/ Ügyfél keresése	Kattintsunk az "Ügyfél keresése" linkre, így ellenőrizzük, hogy az előbb rögzített ügyfél valóban rögzítve lett-e	Az "Ügyfél keresés" képernyőre lépünk	
	Ügyfél keresés képernyő	Vezetéknév: Pallay		
	Keresés	Kattintsunk a "Keresés" gombra	Az ügyfél megjelenik egy táblázatba a megadott adatokkal	
	Vissza a főoldalra	Kattintsunk a "Vissza a főoldalra" linkre	A főoldalra lépünk	
			Az adatbázisba szerepel a felvett ügyfél és a felvett két számla	Az adatbázisba szerepel a felvett ügyfél és a felvett két számla

A funkcionális tesztesetekhez tartozó forgatókönyvek ugyan azokat az adatokat tartalmazzák, mint a JUnit forgatókönyvek.

6.4 Regressziós teszt

Ez a tesztelés a változtatások után kerül alkalmazásra. Az elvégzett módosítások után a fő kérdés, hogy mit kell újra tesztelni? Ha mindent újra tesztelünk az nagyon költségigényes. Ha ad hoc módon az érzéseinkre hagyatkozunk, akkor nem teszteltünk elég alaposan. Akár egy utasítás megváltoztatása is nagy költségeket okozhat. Megfelelő teszteszköz nélkül lehetetlen megállapítani, hogy mit kell újra tesztelni. A regressziós tesztelésnek két fő feladata van. Az egyik meghatározni azokat az output változókat, melyekre a módosítás nem hatna, de mégis ezt teszi, a másik, pedig tesztelni a megváltozott illetve új részeket.

Új funkció bekerülésének tesztelésére egy régi rendszerben szükségünk van a régi tesztesetekre, valamint új tesztesetek megadására. A nem várt hatások tesztelése nehéz. Egy lehetséges módszer rá az összes teszteset újbóli lefuttatása. Egy másik módszer csak a szükséges tesztesetek futtatása. Ehhez ki kell válogatni a megfelelő teszteseteket, ez viszont sok adminisztrációval jár.

A *Regressziós tesztelés* típusai:

- *Régi hibák újratesztelése*: erre a tesztelésre akkor kerül sor, amikor a régi hibák ki lettek javítva. A teszt célja annak bizonyítása, hogy a hiba megszüntetése nem tökéletes. A régi hibák regressziós tesztjenek célja, hogy a régi programhiba kiküszöbölése ellenére az ismét előkerül
- *Mellékhatás regressziós tesztelés*: ez a rendszer lényeges részeinek az újratesztelése. Célja annak igazolása, hogy ami korábban működött az most a program javítása következtében nem működik
- *Tesztelés forgatókönyv szerint*: kézi tesztelés, melyet rendszerint kezdő tesztelő végez jól meghatározott forgatókönyv szerint, amely tapasztalt tesztelő által lett megírva és melybe a tesztlépések lépésről lépésre le vannak írva
- *Smoke tesztelés*: azzal a céllal készül, hogy a termék valamelyik új modulja nem elég érett a további tesztelésre
- *Explorativ tesztelés*: elvárjuk a tesztelőtől, hogy a munka során tanulja meg a terméket, potenciális piacát, a kockázatait és emlékezzen arra, hogy a korábbi tesztek során milyen problémák léptek fel. Az újabb tesztek melyeket a tesztelő elkészít hatékonyabbak, mert a tesztelő növekvő tapasztalatain alapul

- *Gerilla tesztelés*: Gyors és ártalmas támadás a program ellen. Rendszerint tapasztalt tesztelő végzi. A tesztelő a leghatékonyabb módszereket veti be a program ellen. Amennyiben jelentős problémákat talál, ennek a területnek a költségvetését is újra alakítják, és az egész tesztelési folyamat is módosulhat. Amennyiben nem találnak jelentős problémákat, ettől kezdve az adott területet kevésbé intenzíven tesztelik
- *Forgatókönyv tesztelés:* négy tulajdonsággal jellemezhető
    - A teszt legyen valóságszerű. Tükrözze, amit a felhasználó valóban tenne a programmal
    - A teszt legyen összetett, komplex, foglaljon magában számos tulajdonságot, szempontot, amelynek, a programnak meg kell felelnie
    - A teszt legyen könnyen és gyorsan végrehajtható, amelyből megállapítható, hogy a program átment-e a vizsgán vagy sem
    - A tulajdonosok, érintett felek bizonyára erőteljesen amellett érvelnek majd, hogy javítsák ki a programhibákat, ha ilyeneket találnak
- *Telepítés tesztelés*: a tesztelés során a program különböző, megengedett körülmények között és eltérő, megengedett rendszereken telepítésre kerül. Ellenőrzésre kerül, hogy a lemezen mely fájlok adódnak hozzá a meglévőkhöz, illetve melyek módosulnak a telepítés során. Működik-e a telepített program. Mi történik a telepítés megszüntetése (uninstall) után?
- *Betöltődés tesztelés*: a program vagy rendszer futásának ellenőrzése úgy történik meg, hogy közben több programot is betöltenek, illetve a rendszer erőforrásait erősen igénybe veszik. Igen nagy terhelés mellett a rendszer valószínűleg nem működik, azonban ennek a folyamatnak az elemzése rávilágíthat a rendszer sebezhető pontjaira, és ezeket a tapasztalatokat a fejlesztés során hasznosítani lehet
- *Hosszú ideig tartó tesztelés (ellenállóság, megbízhatóság tesztelése)*: a tesztelést egész éjszaka vagy napokig, hetekig folytatják. Ennek a célja olyan hibák felderítése, amelyek rövid idejű tesztelésnél nem kerülnek napvilágra. Ilyenek például a pointerekkel, memória-kezeléssel, memória szivárgással, túlcsordulással és alulcsordulással, és ezek valamilyen kombinációjával, kölcsönhatásával kapcsolatos vizsgálatok

- *Teljesítmény tesztelés*: rendszerint arra irányulnak, hogy milyen gyors a program annak megállapításához, hogy szükséges-e valamilyen optimalizálás. Ezek a tesztek azonban végül számos más programhibát is előidézhetnek, kimutathatnak. A programban a fejlesztés különböző szakaszai között tapasztalt jelentős teljesítményváltozás valamilyen programozási hibára hívhatja fel a figyelmet
- *Kiértékelésre alapozott technikák*: arra irányulnak, hogy milyen módon közöljék a program helyes vagy hibás működését. A kiértékelésre alapozott technikák között leírják azokat a módszereket, amelyek alkalmasak a program helyes vagy hibás működésére vonatkozó megállapítások, következtetések megtételéhez. Ugyanakkor nem mondják meg, hogy hogyan kell magát a tesztelést végrehajtani, hogyan kell adatokat gyűjteni a rendszerről. Arról tájékoztatnak, hogy amennyiben tudunk adatokat gyűjteni, hogyan vonjuk le ezek alapján a következtéseket
- *Önellenőrző adatok*: a tesztelés során az általunk használt adatfájlok alkalmasak lehetnek a kimenő adatok helyességének megállapítására
- *Tesztelés a lementett eredményekkel való összehasonlítás révén*: rendszerint, de nem mindig, automatizált regresszió tesztelés során összehasonlításra kerülnek a jelenlegi eredmények a korábbi, akár múlt heti eredményekkel. Amennyiben a múlt héten az eredmények jók voltak, viszont most más eredmények jöttek ki, ez új problémára hívhatja fel a figyelmünket

6.5 Felhasználói átvételi teszt (UAT – User Acceptance Testing)

Sikeres projektzárás csak az átvételi teszt után következhet.
Az átvételi teszt során általában a felhasználó által végzett, a teljes rendszerre kiterjedő teszt, amelynek eredményeképp a felhasználó elfogadja és átveszi a rendszert.
Az elfogadási tesztnek két fontos feladata van:

- a projekt során korábban alkalmazott tesztelési módszertől eltérő módon, a végfelhasználó szemszögéből tesztelni a rendszert

- a rendszer minőségére vonatkozó megbízható becslést adni. A becslés során az 1000 programutasításban a szoftver átadások meglévő hibák számát adjuk meg. Ez 1-4 közötti érték esetén megfelelő, e felett nem

Átvételi teszt során a teendők a következők:

- a rendszer funkcióinak kellő szintű megismerése
- tesztelési terv kidolgozása, a tesztelés mindig fekete doboz teszt, azaz a program struktúráját ismeretlennek tekintjük
- az elfogadási teszt elvégzése:
    - hibajelentés,
    - a tesztelési idő mérése, hibák megtalálási időpontjának rögzítése,
    - az elfogadási teszt kiértékelése,
    - a rendszer megbízhatóságának becslése

7 Tesztelés dokumentálása

Mint már írtam korábban is nagyon fontos a dokumentumok készítése.

Teszterv: Első sorban teszttervet kell készíteni, mely leírja a tesztelésbe bevont programelemeket, a tesztelési célokat és az alkalmazni kívánt tesztelési módszereket, eszközöket. Ismerteti a teszteléssel kapcsolatos tevékenységeket, azok felelőseit, fázisainak határidejét és a tesztelés során létrehozandó dokumentumokat. Rögzítésre kerül még a tesztkörnyezet (hardver es szoftver).

Jegyzőkönyvek: A tesztekről jegyzőkönyveket kell készíteni. Ezeket általában excel fájlokba írjuk, vagy Word dokumentumokba. A jegyzőkönyvek tartalmazzák a forgatókönyvek végrehajtásának menetét, az egyes lépések sikerességét vagy sikertelenségét. Az JUnit-alapú jegyzőkönyvek nyelve vegyes (angol és magyar).

Teszt specifikáció: részletezik a teszttervben megadott módszereket, tesztelendő jellemzőket és definiálják egy adott specifikációhoz tartozó teszteseteket.

Teszteset, tesztforgatókönyv: (a funkcionális tesztelésnél és a egység tesztelésnél definiáltam)

Hibaleírás: a tesztesetekben az elvárt eredményektől eltérő eredmények leírása.
Tesztelési jelentés: összesíti az átvételi teszt elvégzésekor tapasztaltakat és a tesztelés eredményét.
Tesztelési napló: rögzíti a teszt specifikációk alapján végrehajtott tesztelést, a tesztkörnyezetet és a tesztelés eredményét (akár sikeres akár sikertelen a végrehajtás).

7.1 Hibajelentés és hibakezelés:

A leszállított kész szoftveren (iterációs fázisterméken) a tesztelési tervben meghatározott fázisok menete szerint kell végrehajtani a tesztciklusokat. A tesztelés során felfedezett hibákat („incidenseket") a fejlesztőkkel közösen kell elemezni. Amennyiben az analízis bizonyítja, hogy tényleges szoftverhibáról van szó, hibajegyet kell felvenni, és továbbítani a fejlesztők felé. A fejlesztőkkel való előzetes egyeztetés kétszeresen is visszafizetődik: a fejlesztők a kapott hibajegyet nem vitatják, ha látják rajta fejlesztőtársuk jóváhagyását, valamint a hibajavítás során a hibajelenség reprodukcióját is könnyebben meg tudják oldani.
A hibák jelzésére és követésére a gyártó hálózatában üzemelő rendszert használjuk. A tesztek során felmerülő hibák esetén a teendők az alábbiak szerint alakulnak:

1. A tesztelő jelzi a hibát a fejlesztővezetőnek, közölve annak részletes leírását, körülményeit, az esetleges képernyőmetszeteket, valamint a modult, ahol előjött a hiba.
2. A fejlesztési vezető tovább küldi a hibát valamelyik fejlesztőnek (ha lehet annak, aki az adott modulban fejleszt). A fejlesztő elemzi a hibát, megpróbálja reprodukálni. Ha a hiba reprodukálása nem sikerül, akkor visszaküldi azt a tesztelőnek további információkért. Végül megerősíti a hiba tényét,.
3. Amennyiben a hiba léte megerősítést nyert, a fejleszto ütemezi annak kijavítását.
4. Ha a hiba javítása megtörtént a fejlesztéső a hibabejelentő rendszerben jelzi a hiba kijavításának tényét és visszaküldi a hibát a tesztelőnek.
5. A tesztelő a hiba javítását tartalmazó forráskódváltozatból összeállított új verzión megismétli a teszteket, és amennyiben a korábban hibás tesztek sikeres végrehajtódnak, lezárja a hibajegyet.

A funcionális teszteket időnként a vezető fejlesztő kérése, sok hiba közel egyidejű javítása és nagyobb munkadarabok készrejelentése esetén – regressziós tesztelési céllal is végrehajtjuk.

7.1.1 Hibajelentésre alkalmas eszközök

Bugzilla

A *Bugzilla* rendszert eredetileg a Netscape Communications fejlesztette a Netscape Navigator böngésző fejlesztésének támogatásához. Napjainkra de-facto szabvánnyá nőtte ki magát a hibakövető szoftverek piacán. Az ilyen alkalmazások lehetővé teszik egyének vagy fejlesztőcsoportok számára, hogy a kijavítandó hibákat és a kérelmezett funkciókat nyilvántartsák.

A *Bugzilla* egy jól kiforrott alkalmazás, sok nagyszerű funkcióval rendelkezik. Többek között:

- hatékony keresés
- felhasználók által állítható e-mail beállítások
- teljes hibajegy történet
- hibajegyek függőségi viszonyának követése és grafikus ábrázolása
- dokumentumkezelés
- Web, XML, e-mail és konzol interfészek
- testre szabható és lokalizálható felhasználói felület
- sokféleképpen konfigurálható

A *Bugzilla* rendszert könnyen adaptálni lehet bármilyen feladat követésére. Nyílt hibakövető rendszerként lehetővé teszi, hogy a gyártók kapcsolatba kerüljenek az ügyfeleikkel és viszonteladóikkal, és hatékonyan kommunikálhassanak a problémákról az adatkezelési láncon keresztül. A *Bugzilla* segítségével nő a produktivitás, és lehetővé válik a fejlesztők egyéni munkájának nyomon követése.

Savane

Egy másik rendszer mely alkalmas hibabejelentésre, az a *Savane*, mely a Gna terméke. PHP és Perl nyelvekben íródott, Apache szerveren és MySQL adatbázis szerveren fut. A rendszer egy feladat kezelő, mely alkalmas feladatok és hibák karbantartására.
Fő funkciói:

- hatékony keresés
- felhasználók által állítható e-mail beállítások
- teljes hibajegy történet
- testre szabható felhasználói felület
- robusztus és stabil adatbázisháttér
- sokféleképpen konfigurálható

A fejlesztő cég telepíti az eszközt. Ezután az alkalmazás fejlesztésében résztvevő személyek regisztrálnak a *Savane*-re.
Az eszközben lehetőség van feladatok (task-ok) illetve hibák (bug-ok) rögzítésére, annak kiosztására és azok életciklusának figyelésére.
A rendszerbe való bejelentkezés után a felhasználónak lehetősége van megtekinteni a neki szánt task-okat es bug-okat a *My Incoming Items* funkció segítségével. Ezt kiválasztva a még meg nem oldott, vagy tovább nem adott feladatok, illetve hibák kilistázódnak beérkezési sorrendbe.
Például egy hibát kiválasztva bővebb információkat kapunk róla. Látható azon a felhasználó azonosítója, aki elsőként kiosztotta a hibát, a kiosztás dátuma, a javítás kezdetének és befejezésének dátuma, súlyossága, fontossága, állapota, a hiba részletes leírása, a hozzá fűzött megjegyzések, csatolt fájlok, history (kitől indult a hiba, kiknek lett elküldve, kik küldték tovább). A felhasználó a hiba kezelése után annak állapotát beállítva tovább küldheti azt, vagy lezárhatja.
Új hiba rögzítése esetén meg kell adni annak státuszát, fontosságát, a felhasználót, aki kijavíthatja a hibát, a hiba rövid megnevezését, bővebb leírását, csatolt fájlokat. A hibaküldésre kerül a *Submit* gomb lenyomásával.

7.1.2 A megtalált hibák kezelése

Minden hiba, ami valamilyen szempontból nem felel meg az elvárásoknak, követelményeknek. Ezeket rögzítjük és jelentjük.

Egy hiba adatai:

- azonosító (automatikusan generált egyedi kulcs)
- tárgy (rövid leírás)
- szoftver verziószáma
- konfiguráció (operációs rendszer, hardver, egyéb szoftverkörnyezet)
- a hiba részletes leírása
- reprodukciós lépések
- a szükséges csatolt állományok
- fontossági besorolás (milyen fontos kijavítani)
  például:
  - 1-Wish
  - 3-Minor
  - 5-Normal
  - 7-Important
  - 8-Blocker
  - 9-Security
- súlyossági besorolás (mekkora problémát okoz a hiba)
  például:
  - 1-Later
  - 3-Low
  - 5-Normal
  - 7-High
  - 9-Immediate
- terület/alterület

7.1.3 A hiba életciklus

A tesztelő, aki megtalálta a hibát elküldi azt a területért felelős vezető fejlesztőnek. A vezető fejlesztő átadja a munkát az alterület, modul felelősének. A hiba kijavítása után a megoldás visszakerül a tesztelőhöz, aki leellenőrzi azt.

A hibák lehetséges státuszai:

- Non: semleges
- Fixed: javítva
- Wont Fix: nem javítjuk (alapvető probléma erőforráshiány)
- Works For Me: a hiba a fejlesztőnél nem jön elő, a funkció jól működik
- Ready For Test: tesztre kész
- In Progress: folyamatban
- Postponed: elhalasztva (kockázatkezelés)
- Confirmed: igazolt
- Need Info: további információk szükségesek
- Duplicate: duplikált, már létező hiba
- Invalid: nem reprodukálható

A hiba lezárul, ha hiba gazdája elégedett a javítással, reaktiválódik, ha elégedetlen. A lezárt hiba is reaktiválható regressziós tesztek vagy további ellenőrzések során.

7.2 Tesztelési folyamat minősége

Általában elmondható, hogy a tesztelési folyamat minősége akkor jó ha:

- a hibák a fejlesztés minél korábbi szakaszában derülnek ki és kerülnek javításra
- a kódlefedettség mérőszáma megfelelő (80% felett van)
- a tesztesetek sikeresen végrehajtódtak
- a felmerülő hibák javításra kerültek és átmentek az újbóli teszteléseken
- az elkészült termék és a műszaki folyamat az elfogadási kritériumoknak megfelel

- a megrendelői reklamációk száma csekély a rendszer átadása utan, a megrendelő elégedett a rendszer működésével.

8 Összefoglalás

Diplomamunkám megírása során sikerült mélyebb betekintést nyernem az alkalmazások tesztelésének folyamatába, mely végigkíséri az egész alkalmazás fejlesztés folyamatát. Különböző tesztelési módszereket és eszközöket tanulmányoztam, melyek megkönnyítik a tesztelést. A tesztelési eszközök többségét sikerült ki is próbálnom konkrét példákon, kisebb alkalmazáson.

Az egységtesztelésen belül a JUnit egységtesztelő keretrendszerről írtam. Az egységtesztek kódfedési mértékét meg kell határozni, ehhez több kódfedési eszközt is kipróbáltam, ezek közül részletesen az EclEmma nevű eszközzel foglalkoztam. Ez egyszerű példán keresztül sikerült megcáfolnom a 100%-os kódfedés jelentését. Automatikus JUnit generáló eszközöket is kerestem, sikerült is találnom egy ingyenes eszközt a JUnit Factory-t.

A funkcionális tesztelésen belül a Selenium nevű tesztelő eszköz bemutatása mellett döntöttem. Ez az eszköz elnyerte tetszésem. Létezek más funkcionális tesztelő eszközök, szeretnék még egy néhányat kipróbálni, mint például a WinRunner nevű eszközt. Ehhez sajnos nem sikerült most hozzáférnem, mert egy fizetős eszköz és nincs neki próbaverziója.

A tesztelési módszerek és eszközök után a tesztelés dokumentálásának fontosságáról írtam. Részletesen kitértem a tesztesetekhez, forgatókönyvekhez készült excel fájlokra. Magam is ledokumentáltam két tesztesetet.

Diplomamunkám végén a hibákkal foglalkoztam, azok kezeléséről, életciklusáról. Találtam hibajelentő eszközöket, ebből az egyiket sikerült részletesebben megismernem.

Az elkövetkezőkben szeretnék továbbra is a teszteléssel foglalkozni. Regressziós tesztelési eszközt nem sikerült még tanulmányoznom, automatikus teszteset generáló eszközt sem, pedig azok megkönnyíthetik a tesztelők munkáját, hisz egy komplex rendszerre nem könnyű jó teszteseteket írni.

9 Irodalomjegyzék

R. Binder: *Testing Object-Oriented Systems*, Addison-Wesley, 1999

David Astels: *Test-Driven Development*, Prentice Hall PTR, 2003

Erick M. Burke, Brian M Coyner: *Java Extreme Programming Cookbook*, O'Reilly, 2003

JUnit http://www.junit.org

JUnit Factory http://www.junitfactory.com/

EclEmma http://www.eclemma.org/

Selenium http://selenium.openqa.org/

Bugzilla http://www.bugzilla.org/

Savane https://gna.org/

Köszönetnyilvánítás

Ezúton szeretném megköszönni Dr. Juhász István Tanár Úrnak az egyetemi évek alatt nyújtotta segítségét és szakmai támogatását.